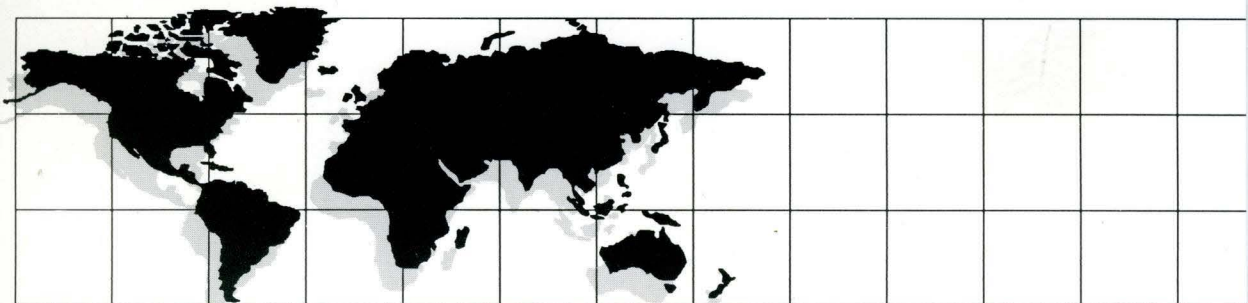




CTOS

Microsoft C

Programming and Installation
Reference

Manual

Volume 2

UNISYS

UNISYS

CTOS[®]

Microsoft C

**Programming and Installation
Reference**

Manual

Volume 2

Copyright © 1992 Unisys Corporation

All rights reserved.

Unisys is a registered trademark of Unisys Corporation.

CTOS is a registered trademark of Convergent
Technologies, Inc., a wholly owned subsidiary of
Unisys Corporation.

Release 6.1

Priced Item

September 1992

Printed in US America

4393 1609-000

The names, places and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintended.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a User Communication Form (UCF) with the CLASS specified as "Documentation", the Type specified as "Trouble Report", and the product specified as the title and part number of the manual (for example, 4393 1609-000).

Shared Resource Processor and SRP are trademarks, and CTOS and SuperGen are a registered trademarks of Convergent Technologies Inc., a wholly owned subsidiary of Unisys Corporation.

Z8000 is a registered trademark of Zilog, Inc.

MAPPER, OFIS, and Unisys are registered trademarks, and CUSTOMCARE and MAPPER are registered service marks of Unisys Corporation. BTOS is a trademark of Unisys Corporation.

PDP-11 and VAX-11 are registered trademarks of Digital Equipment Corporation.

Intel is a registered trademark of Intel Corporation.

Contents

About This Manual	v
-------------------------	---

Section 1. Overview

Compiler and Linker	1-2
NMAKE	1-3
The Programmer's WorkBench	1-5
The CodeView Debugger	1-6
The Browser	1-7
Task Management Service	1-7
Online Help Utilities	1-9
Librarian	1-10
Environment Service	1-11
New Features	1-11

Section 2. Installation and Configuration

System Requirements	2-1
Before You Install	2-2
Preliminary Operations	2-2
Installation Decisions	2-2
Public	2-3
Verbose	2-4
Backup Previous Version	2-4
Save Copy of Backup	2-4
Save Defaults in User File	2-5
Software Destination [sys]	2-5
User Name, CM Configfile Name, Command File Name	2-5
Selecting Software and Directories	2-5
Additional Software	2-8
Installing CTOS Microsoft C	2-8
Installing from Diskettes	2-9
Installing from Tape	2-9
Installing from Your Server Workstation	2-10

Installing Environment Service and Task Management	
Service	2-10
Configuring Your System	2-11
Installing Task Management Service	2-11
Setting up Environment Service	2-12
Sample EnvironmentConfig.Sys	2-13
The CodeView Debugger	2-14
CTOS Libraries	2-14
Video Note	2-14
Increasing the Maximum Number of Open Files	2-15
Increasing File Handles	2-15
Increasing Streams	2-16
Using the Modified Files	2-16

Section 3. Using the Programmer's WorkBench

Starting PWB	3-2
Using Windows and Menus	3-4
Windows	3-4
Menus	3-8
Overview of Menu Commands	3-8
Choosing Menu Commands	3-14
Closing Menus	3-14
Using Shortcut Keys	3-15
Shaded Commands	3-15
Dialog Boxes	3-15
Getting Help	3-18
Menu Help	3-18
Dialog Box Help	3-18
The Help Menu	3-19
Using the Editor	3-19
Moving around in a Source File	3-20
Defining a Block	3-21
Setting Bookmarks	3-22
Setting Anchors	3-23
Searching for and Changing Text	3-24
Creating Macros	3-26
Customizing the Editor	3-27
Changing Editor Settings	3-28
Modifying Keyboard Assignments	3-29
Using Advanced Editor Features	3-30

Compiling, Linking, and Running Programs	3-30
Procedural Overview	3-31
Sample Program	3-34
Saving Source Files	3-35
Setting Compiler Options	3-35
Setting Linker Options	3-39
Saving Compiler and Linker Options as a Build Option	3-41
Creating and Selecting Makefiles	3-43
Notes on Selecting Makefiles	3-45
Selecting a Build Option	3-46
Building Programs	3-47
Running Programs	3-48
Building Programs from Multiple Source Files	3-48
Two Sample Source Files	3-49
Rebuilding IO.Run with Two Source Files	3-50
Quick Reference	3-52
Debugging Programs	3-53
Working with Error Messages	3-54
Using the Browser	3-55
Preparing a Browser Build	3-55
Inspecting with the Browser	3-57
Using the CodeView Debugger	3-61
Preparing a Debug Build	3-62
Starting and Running CodeView	3-64
Debugging IO.c	3-69

Section 4. Using Online Help

Running QuickHelp	4-2
Starting QuickHelp	4-2
Exiting QuickHelp	4-4
Structure of the Help System	4-5
Using Pull-Down Menus and Dialog Boxes	4-7
Pull-Down Menus	4-7
Dialog Boxes	4-15
Navigating through Help Screens	4-15
Navigating with QuickHelp Pull-Down Menus	4-16
Navigating with Hyperlink Commands	4-17
Displaying Hyperlink Topics	4-17
Copying and Pasting	4-18
Programmer's WorkBench	4-19
QuickHelp	4-19

Displaying Keyword Information	4-21
Displaying Error Code Information	4-23
Printing QuickHelp Information	4-24
Configuring a Context Manager Partition	4-25

Section 5. Building Applications

Using the Compiler	5-2
CL Command Form	5-2
Notes on Command Line Options	5-3
CL's MASM Successor Call	5-3
Compiling for Overlays	5-4
Overlay Incompatibilities	5-5
Syntax Errors	5-5
Using the Linker	5-6
Using MLINK	5-7
Command Overview	5-7
Controlling Automatic Linking	5-9
Using an MLINK Response File	5-12
MLINK Defaults	5-12
Using MCBIND	5-14
Cross References between MLINK and MCBIND	
Options	5-15
Creating a Typical Run File	5-19
Startup and Flavor Object Modules	5-21
Specifying Standard Libraries	5-22
Linking a Minimum-Size Run File	5-24
Linking Floating-point Support	5-25
Creating and Naming Combined Libraries	5-25
Startup Object Modules and Heap Architecture	5-27
Allocating Near Heap Memory with malloc() and	
_nmalloc()	5-27
Sizing Near Heap Memory	5-27
Placing Near Heap Memory in DGroup: Three Options ..	5-28
Stack Sharing	5-29
Primary Area with Stack Sharing Overflow	5-30
Primary Area Only	5-30
Sizing and Allocating Memory in Far Heap	5-30
Allocating Far Heap Memory with malloc(),	
_fmalloc(), and halloc()	5-31
LDT-based Segments	5-31
Reserving More Segments with MCMEMORY.Obj ..	5-32
Linking for Real Mode with _huge Memory	5-32

Using the Environment Service	5-33
Notes on Makefiles	5-34
Two Types of Makefiles	5-34
NMAKE with Batch	5-35
Programming Notes	5-36
Using the _plm Keyword	5-36
_plm Keyword Limitations	5-37
Output Redirection	5-39
Input Redirection	5-39
Handling Nameless Structs and Unions	5-40
Manipulating Error Messages	5-40
Binary and Text Modes	5-41
Miscellaneous Function Behaviors	5-42
Functions Not Supported	5-42

Section 6. Optimizing C Programs

How to Specify Compiler Optimization	6-1
Optimizing from Programmer's WorkBench	6-2
Optimizing from the Command Line	6-3
Optimizing from Source Code with Pragmas	6-3
Default Optimizations	6-5
Common Subexpression Elimination	6-5
Dead-Store Elimination	6-6
Constant Propagation	6-6
Customizing Optimizations	6-7
Choosing Speed or Size (/Ot and /Os)	6-8
Generating Intrinsic Functions (/Oi)	6-8
Assuming No Aliasing (/Oa and /Ow)	6-12
Performing Loop Optimizations (/Ol)	6-15
Disabling Unsafe Loop Optimizations (/On)	6-16
Enabling Aggressive Optimizations (/Oz)	6-16
Removing Stack Probes (/Gs)	6-18
Enabling Global Register Allocation (/Oe)	6-18
Enabling Common Subexpression Optimization (/Oc and /Og)	6-20
Achieving Consistent Floating-Point Results (/Op)	6-20
Using the 80186, 80188, or 80286 Processor (/G0, /G1, /G2)	6-21
Optimizing for Maximum Efficiency (/Ox)	6-22
Function Prolog and Epilog Optimization (/Ou and /Oy) ..	6-23

CTOS Linker Optimizations	6-24
Linker Configuration File	6-24
Enabling Far Call Optimization (FarCallTranslation)	6-25
How FarCallTranslation Affects Your Code	6-26
Benefits of FarCallTranslation	6-26
Packing Code (PackCode)	6-28
Choosing Function-Calling Conventions	6-29
The C Calling Convention (/Gd)	6-29
The PLM/Pascal Calling Convention (/Gc)	6-30
The Register Calling Convention (/Gr)	6-30
Argument-Passing Convention	6-31
Return Value Convention	6-32
Stack Adjustment Convention	6-32
Register Preservation Requirement	6-32
Function-Naming Convention	6-33

Section 7. Managing Memory

Pointers and 64K Segments	7-1
Near Pointers	7-2
Far Pointers	7-3
Huge Pointers	7-4
Based Addressing	7-5
Standard Memory Models	7-5
Selecting Memory Models	7-6
Selecting from PWB	7-7
Selecting from the Command Line	7-7
Limitations on Code Size and Data Size	7-8
How to Use The Huge Memory Model	7-8
Null Pointer Sizing Issues	7-9
Mixing Memory Models	7-11
Pointer Problems	7-13
Pointer Conversions	7-15
Rules for Declaring Near, Far, Huge, and Based Variables	7-17
Rules for Declaring Near and Far Functions	7-18

Customizing Memory Models	7-20
Sizing Code Pointers	7-22
Sizing Data Pointers	7-22
Setting Up Segments	7-23
Option /Ad	7-23
Option /Au	7-24
Option /Aw	7-24
Library Support for Customized Memory Models	7-25
Setting the Data Threshold	7-26
Naming Modules and Segments	7-27
Specifying Code and Data Segments	7-29
Using Based Variables	7-30
Advantages of Based Pointers	7-30
Based Pointer Keywords	7-31
Declaring Based Variables	7-32
Variables and Pointers Based on a Segment	
Constant	7-33
Pointers Based on a Segment Variable	7-34
Pointers Based on a Pointer	7-36
Pointers Based on Void	7-38
Pointers Based on a Self Segment	7-38

Section 8. Using the In-Line Assembler

Advantages of In-Line Assembly	8-2
The <code>_asm</code> Keyword	8-2
Using Assembly Language in <code>_asm</code> Blocks	8-4
Instruction Set	8-4
Expressions	8-4
Data Directives and Operators	8-5
EVEN and ALIGN Directives	8-5
Macros	8-5
Segment References	8-5
Type and Variable Sizes	8-6
Comments	8-6
The <code>_emit</code> Pseudoinstruction	8-7
Debugging and Listings	8-7
Using C in <code>_asm</code> Blocks	8-8
Using Operators	8-9
Using C Symbols	8-9
Accessing C Data	8-11
Writing Functions	8-12

Using and Preserving Registers	8-14
Jumping to Labels	8-15
Calling C Functions	8-17
Defining <code>_asm</code> Blocks as C Macros	8-18
Optimizing	8-20

Section 9. Controlling Floating-Point Math Operations

Declaring Floating-Point Types	9-1
Declaring Variables as Floating-Point Types	9-2
Declaring Functions that Return Floating-Point Types	9-4
C Run-Time Library Support of Type <code>long double</code>	9-5
Summary of Math Packages	9-5
Emulator Package	9-5
Math Coprocessor Package	9-6
Alternate Math Package	9-6
Compiling and Linking Floating-Point Options	9-7
In-Line Emulator Option (<code>/FPI</code>)	9-10
In-Line Math Coprocessor Instructions Option (<code>/FPI87</code>) ..	9-11
Calls to Emulator Option (<code>/FPc</code>)	9-11
Calls to Math Coprocessor Option (<code>/FPc87</code>)	9-12
Use Alternate Math Option (<code>/FPa</code>)	9-13
Library Considerations for Floating-Point Options	9-13
In-Line Instructions or Calls	9-13
Compatibility between Floating-Point Options	9-14
Using the <code>NO87</code> Environment Variable	9-15

Section 10. Managing Development Projects with NMAKE

How NMAKE Works	10-2
NMAKE Command Form	10-3
NMAKE Description Files	10-4
Description Blocks	10-4
Escape Sequences	10-7
Wildcards	10-7
Command Modifiers	10-8
Using Control Characters as Literals	10-8
Listing Targets in Multiple Description Blocks	10-10

Comments	10-11
Macros	10-11
User-Defined Macros	10-12
Invoking Macros	10-13
Predefined Macros	10-14
Substitution within Macros	10-17
Inherited Macros	10-19
Precedence among Macro Definitions	10-19
Inference Rules	10-20
User-Defined Inference Rules	10-20
Extension Search Paths	10-21
Predefined Inference Rules	10-22
Precedence among Inference Rules	10-22
Directives	10-23
Pseudotargets	10-26
Command Line Options	10-28
In-Line Files	10-30
NMAKE Operations Sequence	10-31
Differences between NMAKE and MAKE	10-33

Section 11. Creating Help Files with HelpMake

Help Database Content and Format	11-1
Help File Contents	11-2
Help File Formats	11-3
QuickHelp Format	11-4
Minimally Formatted ASCII	11-4
Unformatted ASCII	11-4
Rich Text Format	11-4
Using the HelpMake Command	11-5
HelpMake Options	11-6
Encoding Options	11-7
Decoding Options	11-11
Modifying a Help Database	11-13
Help Text Conventions	11-15
Help Text File Structure	11-15
Local Contexts	11-16
Context Prefixes	11-17
Hyperlinks	11-19

Using Help Database Formats	11-21
QuickHelp Format	11-22
QuickHelp Dot Commands	11-23
QuickHelp Formatting Flags	11-26
QuickHelp Cross References	11-27
Minimally Formatted ASCII Format	11-30
Rich Text Format (RTF)	11-31
Advisor Help Library	11-34

Section 12. Customizing the Programmer's WorkBench

Setting Switches	12-2
Editing from the Options Menu	12-2
Editing TOOLS.INI	12-3
Assigning Keystrokes	12-4
Changing Keystrokes	12-4
Assigning Multiple Keystrokes to One Function	12-4
Disabling Keystrokes	12-5
Limitations	12-5
Writing Macros	12-6
Macro Syntax	12-6
Literal Text	12-6
Definitions	12-7
Passing Arguments to Functions	12-7
Argument Syntax	12-8
Macros Invoking Macros	12-8
Macro Responses	12-8
Macro Arguments	12-9
Macro Conditionals	12-10
Temporary Macros	12-11
Recording Macros	12-12
Writing and Building C Extensions	12-13
Sample Extension	12-14
Writing Mandatory Extension Elements	12-16
Function Prototypes	12-16
Command Table	12-16
Switch Table	12-18
Initializing Function	12-20

Writing Extension Functions	12-20
Receiving Parameters	12-21
Using PWB Functions to Get a File Handle	12-23
Reading a Line from the File	12-24
Writing a Line to the File	12-24
Summary of PWB Functions	12-24
Building, Using, and Debugging Extensions	12-28
Compiling Extensions	12-29
Linking Extensions	12-31
Loading Extensions	12-32
Assigning Keystrokes to Extension Functions	12-33
Debugging Extensions	12-35

Section 13. Debugging C Programs with CodeView

Preparing a Debug Build from the Command Line	13-2
Compiling and Automatic Linking with MLINK	13-4
Compiling and Linking with MCBIND	13-4
Starting the CodeView Debugger	13-6
Understanding the CodeView Debugger Windows	13-8
Adjusting Windows	13-11
Overview of Debugging Techniques	13-11
Controlling Execution	13-12
Continuous Execution	13-12
Selecting and Editing Breakpoint Lines	13-13
Setting Breakpoint Values	13-15
Using Breakpoints	13-17
Single Stepping	13-18
Viewing and Modifying Program Data	13-19
Displaying Variables in the Watch Window	13-19
Displaying Expressions in the Watch Window	13-20
Displaying Arrays and Structures	13-21
Displaying Array Elements Dynamically	13-23
Using Quick Watch	13-24
Displaying Memory	13-25
Displaying Variables with a Live Expression	13-26
Displaying Processor Registers	13-27
Modifying Values of Variables, Registers, and Memory ..	13-27
Memory Window	13-28
Register Window	13-28

Replaying a Debug Session	13-28
Advanced CodeView Debugger Techniques	13-29
Setting Command-Line Arguments	13-29
Multiple Source Windows	13-30
Calling Functions	13-30
Checking for Undefined Pointers	13-30
Printing Selected Items	13-31
Handling Register Variables	13-32
Redirecting CodeView Input and Output	13-32
Customizing the CodeView Debugger from TOOLS.INI ..	13-33

Section 14. Programming with Mixed Languages

Making Mixed-Language Calls	14-1
Library Conflicts	14-3
Language Convention Requirements	14-4
Naming Convention Requirements	14-4
Calling Convention Requirement	14-7
Effects of Calling Conventions	14-8
Parameter-Passing Requirements	14-9
Compiling and Linking	14-11
Compiling with Correct Memory Models	14-11
C Calls to High-Level Languages	14-12
Executing Mixed-Language Calls	14-12
Using the <code>_plm</code> , <code>_fortran</code> , and <code>_pascal</code> Keywords	14-13
C Calls to FORTRAN	14-15
Calling a FORTRAN Subroutine from C	14-15
Calling a FORTRAN Function from C	14-17
C Calls to Pascal and PL/M	14-18
Calling a Pascal or PL/M Procedure from C	14-19
Calling a Pascal or PL/M Function from C	14-20
C Calls to Assembly Language	14-21
Writing the Assembly-Language Procedure	14-22
Setting Up the Procedure	14-23
Entering the Procedure	14-24
Allocating Local Data	14-24
Preserving Register Values	14-25
Accessing Parameters	14-27
Returning a Value	14-30
Exiting the Procedure	14-31

Handling Data in Mixed-Language Programming	14-32
Default Naming and Calling Conventions	14-33
Pascal Conventions	14-33
Numeric Data Representation	14-33
Strings	14-35
C String Format	14-35
FORTRAN String Format	14-36
Pascal String Format	14-36
Array Declaration and Indexing	14-38
Structures, Records, and User-Defined Types	14-39
External Data	14-40
Pointers and Address Variables	14-41
Common Blocks	14-42
Passing the Address of a Common Block	14-42
Accessing Common Blocks Directly	14-43
Using a Varying Number of Parameters	14-43

Section 15. Writing Portable Programs

Hardware Assumptions	15-2
Size of Basic Data Types	15-2
Type char	15-3
Type int and Type short int	15-3
Type float, Type double, and Type long double	15-5
Storage Order and Alignment	15-7
Structure Order and Alignment	15-8
Union Order and Alignment	15-10
Byte Order in a Word	15-10
Storage Alignment	15-12
Reading and Writing Structures	15-12
Bit Fields in Structures	15-12
Order of Assignment	15-13
Size and Alignment of Bit Fields	15-13
CTOS Microsoft C Specific	15-14
Processor Arithmetic Mode	15-14
Pointers	15-15
Casting Pointers	15-16
Pointer Size	15-16
Pointer Subtraction	15-16
Null Pointers	15-17
CTOS Microsoft C Specific	15-18

Address Space	15-19
Character Set	15-20
Character Classification	15-20
Case Translation	15-21
Assumptions about the Compiler	15-22
Sign Extension	15-22
Promotion from Shorter Types	15-22
Bitwise Right-Shift Operations	15-24
Length and Case of Identifiers	15-25
Register Variables	15-26
CTOS Microsoft C Specific	15-26
Functions with a Variable Number of Arguments	15-27
Evaluation Order	15-27
Function and Macro Arguments with Side Effects	15-28
Environment Differences	15-30
Portability of Data Files	15-30
Portability Concerns Specific to CTOS Microsoft C	15-31
CTOS Microsoft C Byte Ordering	15-31

Section 16. Building CTOS III Applications

Creating Multithread CTOS III Applications	16-1
Multithread Programs	16-1
Library Support	16-2
The Multithread C Library LLIBCMT.LIB	16-3
The Multithread Library Compile Option (/MT) ...	16-4
Include Files	16-5
C Run-Time Library Functions for Thread Control ..	16-5
The _beginthread Function	16-6
The _endthread Function	16-7
Writing a Multithread Program	16-7
Sharing Common Resources	16-7
Thread Stacks	16-8
Dynamic Linking with CTOS III	16-8
Overview of Dynamic Linking	16-9
Load-Time and Run-Time Linking	16-9
Application Programs and DLLs	16-10
DLLs and CTOS Microsoft C Run-Time Libraries	16-10
Standalone Dynamic-Link Libraries	16-11
DLLs without C Run-Time Library Functions	16-12

Designing and Writing DLLs	16-13
Floating-Point Math Requirements	16-13
Initialization and Termination Requirements	16-14
Initialization	16-14
Termination	16-16
Making the DLL Re-Entrant	16-16
Global Versus Instance Data	16-17
Using CTOS Microsoft C Keywords	16-18
The _export Keyword	16-18
The _loads Keyword	16-19
Compile Options for Dynamic-Link Libraries	16-19
Compile without Linking (/c)	16-19
Large Memory Model with Separate Stack	
(/ALw)	16-19
Remove Stack Probes (/Gs)	16-20
Specify 80286 Code (/G2)	16-20
Link C Run-Time into Stand-Along DLL (/ML) ...	16-20
Link Executable or DLL with C Run-Time DLL	
(/MD)	16-20
Building DLLs with CTOS Microsoft C	16-21
DLLs with Static C Run-Time Library Functions ...	16-21
Building the DLL	16-21
Building Programs that Call the DLL	16-25
Additional Information	16-27

Appendix A. Migrating to CTOS Microsoft C

Program Changes	A-2
C Functions	A-2
Header Files	A-6
Notes	A-6
Using ctos.lib.h	A-7
Header File Conversions	A-7
Memory Models	A-11
Language Keywords	A-12
Pragmas	A-13
Mixed-Language Programming	A-14
Coding Conventions	A-14
Function Prototypes	A-14
Initialization	A-15
Function and Macro Names	A-15

Compiler Options	A-15
Predefined Macros	A-20
COMDEF OMF Versus PUBDEF Records	A-20
DS Allocation	A-21
Overlays	A-21
C Library Functions	A-22
Math Library Functions	A-30

Appendix B. Command Reference

CL	B-1
Environment Variables Recognized	B-2
Command Line Options	B-2
CodeView	B-10
Summary	B-10
Syntax	B-10
Environment Variables Recognized	B-10
Command Line Options	B-11
CodeView Commands	B-12
Dialog Commands	B-14
Size Specifiers	B-18
Format Specifiers	B-18
Environment Service	B-19
Summary	B-19
Command	B-20
Variable, Examples, and Commands	B-21
Variable Descriptions	B-22
HELPMAKE	B-23
Summary	B-23
Environment Variables Recognized	B-23
Command Line Options	B-23
MLIB	B-28
Summary	B-28
Environment Variables Recognized	B-28
Command Line Options	B-28
MLINK	B-29
Summary	B-29
Syntax	B-29
Environment Variables Recognized	B-30
Command Line Options	B-30

NMAKE	B-38
Summary	B-38
Syntax	B-38
Environment Variables Recognized	B-38
Command Line Options	B-38
Programmer's WorkBench	B-40
Summary	B-40
Environment Variables Recognized	B-40
Command Line Options	B-40
PWBRMake	B-41
Summary	B-41
Environment Variables Recognized	B-41
Command Line Options	B-42
QuickHelp	B-43
Environment Variables Recognized	B-43
Command Line Options	B-43

Appendix C. Implementation-Defined Behavior

Translation	C-2
Diagnostics	C-2
Environment	C-2
Arguments to main	C-3
Interactive Devices	C-3
Identifiers	C-4
Significant Characters without External Linkage	C-4
Uppercase and Lowercase	C-4
Characters	C-5
Multibyte Characters	C-5
Bits per Character	C-5
Character Sets	C-5
Unrepresented Character Constants	C-6
Wide Characters	C-6
Converting Multibyte Characters	C-7
Range of char Values	C-7
Integers	C-7
Range of Integer Values	C-8
Demotion of Integers	C-8
Signed Bitwise Operations	C-9
Remainders	C-9
Right Shifts	C-9

Floating-Point Math	C-10
Values	C-10
Casting Integers to Floating-Point Values	C-10
Truncation of Floating-Point Values	C-11
Arrays and Pointers	C-11
Largest Array Size	C-11
Casting Pointers	C-11
Pointer Subtraction	C-12
Registers	C-12
Availability of Registers	C-12
Structures, Unions, Enumerations, and Bit Fields	C-13
Improper Access to a Union	C-13
Sign of Bit Fields	C-13
Storage of Bit Fields	C-14
Alignment of Bit Fields	C-14
The Enum Type	C-15
Qualifiers	C-15
Access to Volatile Objects	C-15
Declarators	C-15
Maximum Number	C-15
Statements	C-15
Limits on Switch Statements	C-15
Preprocessing Directives	C-16
Character Constants and Conditional Inclusion	C-16
Including Bracketted File Names	C-16
Including Quoted File Names	C-17
Character Sequences	C-18
Pragmas	C-18
Default Date and Time	C-19
Library Functions	C-19
NULL Macro	C-20
Diagnostic Printed by the assert Function	C-21
Character Testing	C-21
Domain Errors	C-22
Underflow of Floating-Point Values	C-22
fmod Function	C-22
Terminating Newline Characters	C-22
Blank Lines	C-23
Null Characters	C-23
File Position in Append Mode	C-23
Truncation of Text Files	C-23

File Buffering	C-23
Zero-Length Files	C-24
File Names	C-24
File Access Limits	C-24
Deleting Open Files	C-24
Renaming with a Name that Exists	C-25
Printing Pointer Values	C-25
Reading Pointer Values	C-25
Reading Ranges	C-25
File Position Errors	C-26
Messages Generated by the perror Function	C-26
Allocating Zero Memory	C-27
abort Function	C-27
atexit Function	C-27
Environment Names	C-28
system Function	C-28
strerror Function	C-28
Time Zone	C-30
Clock Function	C-30

Appendix D. Compiler Limits and Numerical Ranges

Compiler Limits	D-2
Numerical Ranges	D-4

Appendix E. printf/scanf Format Specifiers E-1

Appendix F. Re-entrant Run-Time Library Functions F-1

Appendix G. Sample NMAKE Description Files

Help File G-2
Library Version String G-4
Programmer's WorkBench Extension G-6
Run File G-12
Library G-21
Run Files, System Service, and Libraries G-30

Glossary Glossary-1

Index Index-1

Figures

1-1.	Building Programs in One Step	1-2
1-2.	Building Programs in Two Steps	1-3
1-3.	Building Programs with a Makefile	1-4
1-4.	Building and Debugging Programs from PWB	1-6
1-5.	Task Management Service	1-8
1-6.	Relationships between Help Utilities	1-10
3-1.	Parts of a PWB Screen.	3-5
3-2.	Dialog Box for the Find Command	3-16
3-3.	Define Mark Dialog Box	3-22
3-4.	Setting and Saving Build Options	3-32
3-5.	Creating a Makefile	3-33
3-6.	Building Programs	3-34
3-7.	Dialog Box for the Save As Command	3-35
3-8.	C Compiler Options Menu	3-36
3-9.	C Compiler Debug Options Menu	3-37
3-10.	C Compiler Release Options Menu	3-38
3-11.	Link Options Menu	3-39
3-12.	Link Debug Options Menu	3-40
3-13.	Link Release Options Menu	3-41
3-14.	Build Options Dialog Box	3-42
3-15.	Dialog Box for the Set Program Command	3-43
3-16.	Makefile Dialog Box	3-44
3-17.	Edit Program List Command Form	3-44
3-18.	Set Initial Build Options Dialog box	3-46
3-19.	Browse Options Dialog Box	3-56
3-20.	Browse Menu	3-58
3-21.	Browser Display of Functions	3-61
3-22.	CodeView Options Menu	3-65
3-23.	CodeView Startup Screen	3-66
3-24.	CodeView Screen with Watch and Register Windows	3-67
3-25.	CodeView Screen with Expanded Arrays	3-68

Figures

4-1.	QuickHelp Initial Screen	4-4
4-2.	Global Contents Help Screen	4-5
4-3.	Global Index Help Screen	4-6
4-4.	File Commands	4-8
4-5.	View Commands	4-9
4-6.	Categories Commands	4-10
4-7.	References Commands	4-11
4-8.	Paste Commands	4-12
4-9.	Options Commands	4-13
4-10.	Dialog Box for the Search Command	4-15
4-11.	Sample Index for Keywords	4-22
5-1.	MCBIND Command Form	5-15
5-2.	Sample MCBIND Link	5-20
5-3.	Near Heap Memory Organization Options	5-29
7-1.	Anatomy of Small-Model Program	7-3
10-1.	Typical Description Block	10-5
12-1.	Sample CL Command Form for Building Extensions	12-29
12-2.	Sample CL Command Form for Compiling Extensions	12-30
12-3.	Sample MCBIND Command Form for Building Extensions	12-31
13-1.	Sample CL Command with Linking for a CodeView Build	13-4
13-2.	Sample CL Command without Linking for a CodeView Build	13-4
13-3.	Sample MCBIND Command for CodeView Build	13-5
13-4.	Sample CodeView Startup Screen	13-9
13-5.	Edit Breakpoints Dialog Box	13-14
13-6.	Set Breakpoint Dialog Box	13-16
13-7.	Add Watch Dialog Box	13-19
13-8.	Quick Watch Dialog Box	13-24
13-9.	Memory Window Options	13-25
13-10.	Print Dialog Box	13-31
14-1.	Mixed-Language Call	14-2
14-2.	Naming Convention	14-6
14-3.	C Stack Frame	14-28
14-4.	C String Format	14-35
14-5.	FORTTRAN String Format	14-36
15-1.	Data Alignment in Bit Fields	15-14
16-1.	DLL and Program with Statically Linked C Run-Time Functions	16-12

Tables

2-1.	Software Selection	2-6
2-2.	Additional Software Selection	2-8
3-1.	Parts of a PWB Screen	3-6
3-2.	Command Overview	3-9
3-3.	PWB Dialog Box Features	3-16
3-4.	Browse Commands	3-59
4-1.	Hyperlink Commands	4-17
4-2.	Ways to Display Unmarked Hyperlink Topics	4-18
5-1.	Compiler Options Used for Overlays	5-4
5-2.	CL Compiler Options Incompatible with Overlays	5-5
5-3.	MLINK Command Fields	5-8
5-4.	MLINK Defaults Correlated to MCBIND Fields	5-13
5-5.	Correlating MCBIND and MLINK Options	5-16
5-6.	Correlating CTOS Linker Config File and MLINK Options	5-18
5-7.	Startup Object Modules	5-22
5-8.	Flavor Object Modules	5-22
5-9.	Standard Libraries	5-23
5-10.	Combined Libraries	5-26
5-11.	Unsupported Library Functions	5-43
6-1.	Processor Compatibility	6-22
6-2.	Processor Clock Cycles for Calling Sequence	6-27
6-3.	Register Candidates	6-31
7-1.	Memory Models	7-6
7-2.	Memory Model Options and Libraries	7-7
7-3.	Addressing Declared with Microsoft Keywords	7-12
7-4.	Start-Up Routines for Customized Memory Models	7-26
7-5.	Segment-Naming Conventions	7-28
7-6.	Keywords	7-32

Tables

9-1.	Floating-Point Types	9-2
9-2.	Lengths of Exponents and Mantissas	9-2
9-3.	Range of Floating-Point Types	9-3
9-4.	Summary of Floating-Point Compiler Options	9-8
9-5.	Floating-Point Options and Libraries	9-9
10-1.	Command Modifiers	10-8
10-2.	Predefined Macros	10-14
10-3.	Predefined Inference Rules	10-22
10-4.	Directives	10-23
10-5.	Directive Operators	10-25
10-6.	Pseudotargets	10-27
10-7.	NMAKE Options	10-28
11-1.	Options for Encoding	11-7
11-2.	Options for Decoding	11-11
11-3.	Standard h. Contexts	11-17
11-4.	Microsoft Product Context Prefixes	11-19
11-5.	QuickHelp Dot Commands	11-24
11-6.	Formatting Flags	11-26
12-1.	Callable PWB Functions	12-25
13-1.	List of CodeView Windows	13-8
14-1.	Language Equivalents for Routine Calls	14-3
14-2.	Methods for Passing Parameters	14-10
14-3.	Parameter-Passing Defaults	14-11
14-4.	Register Conventions for Simple Return Values	14-30
14-5.	Default Naming and Calling Conventions	14-33
14-6.	Equivalent Numeric Data Types	14-34
14-7.	Equivalent Array Declarations	14-39
15-1.	Size of Basic Types in CTOS Microsoft C	15-3
15-2.	Size of Generic Pointers	15-18
15-3.	Default Pointer Sizes	15-19
15-4.	Byte Ordering for Short Types	15-32
15-5.	Byte Ordering for Long Types	15-32
16-1.	Multithread Options	16-4
16-2.	Stand-Alone DLL Files	16-21
16-3.	Linking Files Needed	16-26

A-1.	Documented BTOS C Functions Without Substitutes	A-3
A-2.	Documented BTOS C Functions With Substitutes	A-4
A-3.	Undocumented BTOS C Functions With Work-arounds	A-5
A-4.	BTOS C and CTOS Microsoft C Header Files	A-8
A-5.	BTOS C and CTOS Microsoft C Extended Keywords	A-13
A-6.	BTOS C and CTOS Microsoft C Compiler Options	A-16
A-7.	BTOS C Function Status in CTOS Microsoft C	A-22
A-8.	BTOS C Math Function Locations in CTOS Microsoft C	A-30
C-2.	Source and Execution Character Sets	C-6
D-1.	Limits Imposed by the C Compiler	D-2
D-2.	Program Limits at Run Time	D-3
D-3.	Data Ranges Defined in LIMITS.H	D-4
D-4.	Numerical Values Defined in FLOAT.H	D-6
E-1.	printf and scanf Format Specifiers	E-1

This Page is Blank
Do Not Print

Section 12

Customizing the Programmer's WorkBench

The Programmer's WorkBench (PWB) provides a highly extensible development platform. Within PWB you can modify the Editor by changing switch settings and keystroke assignments, and extend functionality through macros and C-language extensions.

This section explains how to customize PWB using four methods:

- Setting Switches
- Assigning Keystrokes
- Writing Macros
- Writing and Building C Extensions

While this section explains customization methods, it does not document all customizable features of PWB. For complete information, refer to online Help. To understand the information in this section, you must be familiar with basic PWB operations and terminology. For an introduction on how to use PWB, refer to Section 3, Using the Programmer's WorkBench. For a list of PWB startup options, refer to Appendix B, Command Reference.

Note: *If you are familiar with the DOS version of Microsoft C, note that for CTOS the ALT key maps to COPY, and CRTL maps to CODE. For keyboards without ALT, use COPY. In addition, pseudofiles for the DOS version are enclosed with angle brackets < >. For CTOS the pseudofiles are enclosed in parentheses (. For example, <assign> maps to (assign).*

Setting Switches

PWB has a number of switches (user-configurable options) that control features such as key assignments and key functions. All switches have names and can be assigned values.

You can set PWB switches two ways:

- Edit Editor Settings from the Options Menu
- Edit TOOLS.INI

Editing from the Options Menu

When you choose Editor Settings from the Options menu, PWB displays the (assign) pseudofile in the current window. PWB constructs pseudofiles dynamically, which means they exist in memory only. The (assign) pseudofile lists all current PWB switch settings.

To change a switch, edit the line where it appears. For example, the **vscroll** switch specifies the number of lines PWB scrolls vertically. The default is 1. To change the default, find this line:

```
vscroll:1
```

Change 1 to 3 and move the cursor to another line. PWB now scrolls three lines at a time.

To indicate a legal change, PWB highlights the line. To indicate an illegal change, PWB displays an error message. That is, changes occur immediately.

If you do not explicitly save changes before exiting PWB, they disappear. To save changes, save (assign) from the File menu like any other file. When saving this file, PWB updates TOOLS.INI.

You can also use this method for more elaborate customizations, such as writing macros, explained in detail in this subsection. To insert macros, insert a few blank lines in (assign) and enter new information. Note that lines added or changed affect PWB, but lines deleted have no effect.

Editing TOOLS.INI

Another way to customize PWB is by editing TOOLS.INI, the initialization file that PWB uses. This method is useful if you customize PWB extensively. With this method, you can create different sections that affect switches, affect files with specific extensions, and affect tools other than PWB.

While the (assign) pseudofile lists every customizable PWB item, TOOLS.INI lists changed items only. That is, values that appear in TOOLS.INI override defaults, and implement other options not set by defaults such as extensions. Settings that follow the [PWB] tag affect PWB every time it starts.

For example, suppose you set vscroll to 3 in the (assign) pseudofile and saved the change, but did not otherwise customize PWB. These lines now appear in TOOLS.INI:

```
[PWB]
vscroll:3
```

The screen now scrolls 3 lines at a time.

You can create sections in TOOLS.INI that PWB reads under certain circumstances only. For example, you can also create a section for files that have specific extensions, so PWB uses specific instructions when loading files with these extensions. Specifically, TOOLS.INI can contain a section for C source files that begins with the tag:

```
[PWB-.c]
```

For assembly-language (.ASM) source files, TOOLS.INI can contain a section that begins with the tag:

```
[PWB-.ASM]
```

When loading files with these extensions, PWB first reads the appropriate section of TOOLS.INI for specific instructions. For each file type, you can use different sets of macros and other customizations.

Since several tools use TOOLS.INI, the file can also contain information unrelated to PWB. If you customize other tools, TOOLS.INI lists separate sections for each tool. Each section begins with a tag that shows the tool's name in square brackets: [PWB], [NMAKE], and so forth.

Assigning Keystrokes

PWB allows you to assign editing functions to keystrokes. However, reassigning keystrokes does not change the PWB graphic interface on the pull-down menus.

This subsection covers the following topics:

- Changing Keystrokes
- Assigning Multiple Keystrokes to One Function
- Disabling Keystrokes
- Limitations

Changing Keystrokes

Keystrokes, like switches, appear and can be changed in the (assign) pseudofile by selecting Key Assignments from the Options menu. For example, to assign the tab key to the Shift-TAB keystroke, find this line (default):

```
tab:tab
```

Change it to:

```
tab:shift+tab
```

Now, pressing SHIFT-TAB performs the tab key function.

Assigning Multiple Keystrokes to One Function

You can assign multiple keystrokes to the same function. For example, these tags assign two keystrokes to the tab key:

```
tab:shift+tab  
tab:shift+F5
```

Now, pressing either SHIFT-TAB or SHIFT-F5 performs the tab key function.

Disabling Keystrokes

Occasionally, you may want to unassign, or disable, a keystroke. To do so, assign the **unassigned** function to the keystroke. For example,

```
unassigned:shift+tab
```

disables SHIFT-TAB. When you press an unassigned key, PWB signals an error.

Limitations

Keystroke assignments have three limitations:

- You cannot reassign a keystroke that PWB is using for a menu. For instance, if SHIFT-F5 pulls down the File menu, PWB ignores attempts to reassign SHIFT-F5.
- You cannot reassign COPY plus the number keys 1 - 9 (COPY-1, COPY-2, and so forth). PWB reserves these keystrokes for the file history menu items.
- Each keystroke can invoke one function only. If you mistakenly assign a key to more than one function, PWB uses the most recent assignment. For example,

```
home:shift+tab  
setfile:shift+tab
```

assigns the SHIFT-TAB keystroke to two different functions, home and setfile. However, the second assignment overrides the first, assigning SHIFT-TAB to setfile. And as shown here, more recent assignments appear lower in the list.

Writing Macros

The fastest way to create new PWB editing functions is with macros. The new function can be as simple as one that inserts a word, or as complex as one that invokes other macros.

To explain macros, this subsection covers these topics:

- Syntax
- Responses
- Arguments
- Conditionals
- Temporary Macros
- Recordings

Macro Syntax

A macro can contain any combination of PWB functions, literal text, macro operators, and extension functions. You can define up to 1,024 macros.

This subsection covers the following topics:

- Literal Text
- Definitions
- Passing Arguments to Functions
- Argument Syntax
- Macros Invoking Macros

Literal Text

Literal text is anything inside double quotes. Inside literal text, you can represent a double quote as \" and a backslash as \\. Text is case sensitive inside quotes and case insensitive outside them.

The following macro comments out a line of C source code:

```
comment:=begline "/* " endline " */"  
comment:alt+c
```

The first line names the macro and indicates what it does. The begline and endline editor functions move the cursor, while text inside quotes appears at the cursor position. The second line assigns the keystroke CODE-C to the macro.

Definitions

A macro definition must fit on one logical line. To continue a definition on the next line, use the backslash (\). For example, the definition

```
comment:=begline "/* " endline " */"
```

could be written as

```
comment:=begline  \  
"/* " endline  \  
" */"
```

Note the extra space before each backslash. To put a space between the end of one line and the beginning of another, precede the backslash with two spaces.

Passing Arguments to Functions

To pass arguments to functions, use the arg function. For example, the following macro passes the argument 15 to the plines function (which scrolls text down):

```
movedown:=arg "15" plines
```

Because arg precedes the literal text, the text does not appear on the screen. Instead, the system passes 15 as an argument to the next function, plines. Thus, the macro scrolls the current text down 15 lines.

Argument Syntax

Arguments can use regular expression syntax. For example, in the line:

```
endword:=arg arg "( !.!$!\\:;!\\)!\\(!,)" psearch
```

the `arg arg` sequence directs the `psearch` function to treat the text argument as a regular expression search pattern. This search pattern tells PWB to search for the next period, end of line (\$), colon, semicolon, closed parenthesis, open parenthesis, or comma.

For details about regular expressions, refer to online Help.

Macros Invoking Macros

A macro can also invoke other macros. In the following example:

```
lcomment:= "/* "  
rcomment:= " */"  
commentout:=begline lcomment endline rcomment  
commentout:copy+z
```

the `commentout` macro invokes the previously defined macros `lcomment` and `rcomment`.

Macro Responses

Before executing, some PWB functions prompt you for confirmation. For example, the `meta exit` function (quit without saving) normally asks if you really want to exit without saving. These type of prompts always require yes (y) or no (n).

However, when you invoke such a function in a macro, the function assumes an answer of yes and does not prompt for confirmation. For example, the macro definition

```
quit:=meta exit  
quit:copy+x
```

invokes `meta exit` when you press `COPY-X`. But because the `meta exit` function is invoked from a macro, PWB exits without asking for confirmation.

To restore normal prompting or change the default responses, use the following operators:

Operator	Description
<	Prompts for confirmation; if not followed by another < operator, prompts for all further questions
<y	Assumes a response of yes
<n	Assumes a response of no

A response operator applies to the function immediately preceding it. For instance, to restore the usual prompt, add the < operator to the quit macro definition:

```
quit:=meta exit <
quit:copy+x
```

Now, before you exit, the macro prompts for a response.

Macro Arguments

If you enter an argument then invoke a macro, PWB passes the argument to the first function in the macro that takes an argument. For example, the macro:

```
tripleit:=copy paste paste
```

invokes the functions **Copy** and **Paste**.

If you highlight text and invoke the macro, the macro passes your highlighted argument to the **Copy** function, which copies it to the clipboard. The macro then invokes **Paste** twice, which inserts two copies of the highlighted text.

You cannot pass more than one argument from PWB to a macro, even if the macro invokes more than one function that can accept an argument. The argument always goes to the first function in the macro that takes an argument.

You can prompt for input inside a macro and pass the input as an argument using the prompt function, as shown here:

```
newfile:=arg "Next file: " prompt setfile <
newfile:copy+n
```

In this example, the `newfile` macro prompts for a file name and then switches to the file specified. The sequence `arg "Next file: "` passes a text argument to `prompt`, which displays the text on the dialog line and waits for input. When invoked, PWB passes the input as a text argument to the `setfile` function, which switches to that file. For more information on the `prompt` function, refer to online Help.

Macro Conditionals

Macros can act according to True and False conditions. Such macros take advantage of the fact that PWB editing functions generally return values whether or not they are successful. These functions return TRUE if successful (nonzero), and return FALSE if unsuccessful (zero).

Macros can use four conditional operators:

Operator	Description
<code>:>label</code>	Defines a <i>label</i> that can be targeted by other operators
<code>=>label</code>	Jumps to <i>label</i>
<code>+>label</code>	Jumps to <i>label</i> if the previous function returns TRUE
<code>->label</code>	Jumps to <i>label</i> if the previous function returns FALSE

For example, the following macro moves the cursor to the left margin of the editing window:

```
leftmarg:=:>leftmore left +>leftmore
```

Here, `leftmarg` invokes the `left` function repeatedly (jumping to the label `leftmore`) until it returns FALSE, which indicates the cursor reached the left margin.

The label must also appear immediately after the conditional operator with no spaces in between. A conditional operator without a label exits the macro immediately if the condition is true. If the condition is false, the macro continues execution.

For example, the macro:

```
turnon:=insertmode +> insertmode
```

turns on insert mode whether insert mode is currently on or off.

If insert mode is off, the first invocation of `insertmode` toggles the mode on and returns `TRUE`, causing the `+>` operator to terminate the macro. If insert mode is currently on, the first invocation of `insertmode` turns insert mode off and returns `FALSE`. The macro then invokes `insertmode` a second time, turning insert mode on.

Temporary Macros

Occasionally, you may want to create a macro for the current session only. To create a temporary macro, use the `assign` function.

For example, the following steps create a macro described earlier in this section, the one that creates C comments.

To create the macro:

1. Press **COPY-A**.
2. Type `comment:=begline "/* " endlne "*/"`.
3. Press **COPY-=**.

To assign the keystroke `COPY-C` to the macro:

1. Press **COPY-A**.
2. Type `comment:Copy+c`.
3. Press **COPY-=**.

The macro is available immediately, but disappears when the current session ends.

Recording Macros

In addition to defining macros, you can record macros. Once you record macros, you can replay the entire sequence by pressing one key or a combination of keys.

To record macros, you implicitly invoke the record function. By default, PWB names the resulting macro **recordvalue**, but you can use other names as well.

To record a macro:

1. Select **Edit, Record On**.
2. Press the keys to record.
3. To end, select **Record On** again.
4. If **recordvalue** is not assigned, assign it to a keystroke as previously described.

When you finish recording, a macro named **recordvalue** executes whenever you press the key assigned in the last step. When you press the key, PWB replays recorded keystrokes. If you do nothing else, the macro is temporary; it disappears when you exit PWB.

To save the macro permanently:

1. Open the <record> pseudofile by pressing **COPY-A**, type <record>, press **F2**.
2. **Copy** the macro definition in <record>.
3. Open your **TOOLS.INI** file, and **Paste** the definition in the **[PWB]** section.

By studying recorded macros, you can learn much about macros and editor functions. One way to study macros is to watch PWB write macros when recording. To watch PWB write macro definitions, open the <record> pseudofile in a second window before recording.

If you save a recorded macro, you will want to name it something other than **recordvalue**, the default.

To override the default, pass the new name as an argument before you start the recording, as shown in these steps:

1. Press **COPY-A COPY-A**.
2. Type the new name.
3. Select **Edit, Record On**.
4. Complete the recording as usual.

You can append commands to an existing macro using the same process. Instead of typing a new name, type an existing name. PWB then appends the recorded commands to the macro instead of replacing it.

You can also make a silent recording that records a series of actions without executing them. Start the recording with a meta record command (press F9 SHIFT-CODE-R). Then complete the recording process as previously described.

Writing and Building C Extensions

An extension is a file that contains one or more user-written functions that can be assigned keystrokes, given arguments, and invoked in macros like other PWB functions. PWB loads extensions at run time.

The ability to load and call user-written functions makes PWB highly extensible. Because they consist of compiled C code, your extensions execute faster than macros, and they can perform jobs that are more complex.

Although extensions contain executable code, they differ from Run files in the following ways:

- Extensions do not contain the usual C start-up code.
- Extensions contain special data structures that describe functions to PWB.
- Extensions are declared in a way so PWB can call and pass arguments to them.
- Extensions can call native PWB functions, and some but not all C library functions.

This subsection explains how to write, build, and invoke PWB extensions, and covers these topics:

- Writing Mandatory Extension Elements
- Writing Extension Functions
- Building and Using Extensions

Note: *The PWB extension interface is designed for C programmers. However, you can write extensions in assembly language or other languages if you simulate the required C memory model (in which SS is not assumed to equal DS).*

Sample Extension

To illustrate how to create extensions, the rest of this section describes concepts as they apply to the following example, Center.c. This extension contains one function, CenterLine, that centers a line or range of lines in the current file.

```
/* CENTER.c: Sample PWB extension */

#define LINE_LENGTH 80          /* Assume 80-column screen */
#include <string.h>              /* PWB extension header file */
#include <ext.h>

/* Function Prototypes */
PWBFUNC CenterLine( unsigned argData,
                    ARG _far *pArg,
                    flagType fMeta );

/* Command Table */
struct cmdDesc  cmdTable[]
={ { "CenterLine", CenterLine, 0, NOARG | LINEARG },
  { NULL, NULL, 0, 0 }
};

/* Switch Table */
struct swiDesc  swiTable[]
={
  { NULL, NULL, 0 }
};

/* Initialization Function */
```

```

void EXTERNAL WhenLoaded( void )
{ DoMessage( "Loading Center extension" );
}

/* Extension (user-written) function */

PWBFUNC CenterLine( unsigned argData,
                    ARG_far *pArg,
                    flagType fMeta )
{ PFILE pFile;
  LINE yStart, yEnd;
  int len;
  char *pBuf, buf[BUFLen];

  /* Get handle to current file */
  pFile = FileNameToHandle( "", "" );
  /* Handle various argument types */
  switch( pArg->argType )
  /* Center current line */
  { case NOARG:
    yStart = yEnd = pArg->arg.noarg.y;
    break;

    /* Center range of lines */
    case LINEARG:
    yStart = pArg->arg.linearg.yStart;
    yEnd = pArg->arg.linearg.yEnd;
    break;
  }
  /* Center current line or range of lines */

  for( ; yStart <= yEnd; yStart++ )

    /* Get a line from current file */
    { len = GetLine( yStart, buf, pFile );

      if( len > 0 )
        /* Center text in this line */
        { pBuf = buf + strstr( buf, "\t" );
          len = strlen( pBuf );
          memmove( buf+(LINE_LENGTH-len) / 2, pBuf, len+1);
          memset( buf, ' ', (LINE_LENGTH - len) / 2 );

          /* Write modified line back to the current file */
          PutLine( yStart, buf, pFile );
        }
    }
  return TRUE;
}

```


Writing Mandatory Extension Elements

Because extensions do not contain `main()` functions, they must contain these elements to work properly:

- Function Prototypes
- Command Table
- Switch Table
- Initializing Function

This subsection explains how to create these elements, and what role they play in making the extension work.

Function Prototypes

To be called by PWB, each extension function must be declared as type `PWBFUNC` and accept the parameters `argData`, `*pArg`, and `fMeta`. The `CenterLine` function in the section of `Center.c` code below follows this model:

```
PWBFUNC CenterLine( unsigned argData,  
                    ARG _far *pArg,  
                    flagType fMeta );
```

The `PWBFUNC` type is actually a macro that evaluates to `flagType _pascal _loadds _far`. The `flagType` return type declares that the function returns either `TRUE` (nonzero) or `FALSE` (zero). Your function should return a value so that it can be used in a macro with conditionals. The modifiers `_pascal`, `_loadds`, and `_far` specify the calling conventions PWB expects editor functions to have.

Command Table

Every extension must contain an array of structures named `cmdTable`. This array provides the resident function names that PWB uses to call the extension's functions.

`cmdTable` is an array of structures of type `cmdDesc` (declared in `ext.h`). Each structure except the last one describes one extension function. The last structure, always of null members and zeros, terminates the array.

For example, the `Center.c` extension has one function named `CenterLine`. Thus, the `cmdTable` array contains two structures, one for `CenterLine`, and one to terminate the table:

```
struct cmdDesc cmdTable[]
={
  { "CenterLine", CenterLine, 0, NOARG | LINEARG },
  { NULL, NULL, 0, 0 }
};
```

Each `cmdDesc` structure in `cmdTable` contains four members:

- Function name
- Pointer to the function
- Reserved item (must be 0)
- Bit Field that lists types acceptable to the function

The first member contains the function name that PWB uses to call the function.

The second member is the PWB function address.

The third member is reserved and always set to zero.

The last member is an integer that contains bitflags. These bitflags represent argument types that your function accepts. You can combine multiple bitflags using the OR (`|`) operator.

For example, the `CenterLine` function can handle an argument of type `LINEARG`, or no arguments (`NOARG`). Thus, it lists the types:

`NOARG | LINEARG`

In addition to these arguments, between fifteen and twenty other types exist. For details, refer to the `Extensions` topic in online Help.

Switch Table

Every extension must contain an array of structures named **swiTable**. This array provides a way to control the functions specified in **cmdTable** by passing variables that users can change at run time. For example, you could configure a numeric switch that specifies how many lines of text to delete for a function that deletes lines of text.

swiTable is an array of structures of type **swiDesc**. Each structure except the last one describes one switch used by a function in your extension. The last structure, always composed of null members and zeros, terminates the array. It is not mandatory to set switches with this array, but the array must always appear in your extension.

For example, the **Center.c** extension does not take switches, so the **swiTable** array contains a terminating null structure only:

```
struct swiDesc swiTable[]
= {
    { NULL, NULL, 0 }
};
```

Each **swiDesc** structure in **swiTable** contains three members:

- Switch name
- Pointer to a data variable or code function that defines the switch
- Flag that indicates one of three switch types

The first member contains the switch name that appears in the Editor Settings Option window. The name can be any text string.

The third member specifies the type of switch to retrieve from the user, and is one of three types: **SWI_BOOLEAN** for TRUE/FALSE conditions, **SWI_NUMERIC** for numerics, or **SWI_SPECIAL** for strings.

The second member is a pointer that behaves differently depending on the switch type specified in the third member. For **SWI_BOOLEAN** and **SWI_NUMERIC**, the pointer points to the switch itself. For **SWI_SPECIAL**, the pointer points to a string-handling function.

For example, the following code creates a numeric switch with the default value 27:

```
static int n = 27;

struct swiDesc swiTable[]
=
{
    { "newswitch", &n, SWI_NUMERIC | RADIX10 },
    { NULL, NULL, 0 }
};
```

In this example, members in the first structure contain the switch name `newswitch`; the pointer `&n`, which points to the variable that contains the switch value; and the switch type `SWI_NUMERIC`.

Also, notice that the third member contains another constant, `RADIX10`. If a switch is type `SWI_NUMERIC`, you must supply a second constant to tell PWB whether to interpret user-assigned values as decimal (`RADIX10`) or hexadecimal (`RADIX16`) numbers.

If the third member is type `SWI_SPECIAL`, the second member of `swiDesc` is a pointer to an additional string-handling function that you write. This function must be of type `int _far _pascal`. Each time the text switch changes, PWB calls your function, passing it the address of the updated string as a `char _far` pointer. The following code stores the updated string in a buffer named `mystring`:

```
char mystring[BUFLen];

int _far _pascal setstr( char _far *ptr )
{ strcpy( mystring, ptr );
}
```

If desired, you can list switches for extension functions separately from other switches. Whenever PWB loads an extension, it looks in `TOOLS.INI` for a section with this form:

```
[PWB-ext]
```

where *ext* is the base name of the extension. If the extension exists, PWB recognizes the settings immediately following the tag. For instance, if your extension `SAMPLE.CXT` uses a numeric switch named `numbills`, you can set `numbills` to the value 66 with:

```
[PWB-SAMPLE]
numbills:66
```

Initializing Function

Every PWB extension must contain a function named `WhenLoaded`, which PWB calls immediately after loading the extension. The `WhenLoaded` function allows you to initialize your functions if required. If your functions do not require initialization, they can return.

The `Center.c` extension uses `WhenLoaded` to display a loading message:

```
void EXTERNAL WhenLoaded( void )
{ DoMessage( "Loading Center extension" );
}
```

`DoMessage` is a PWB function that displays a message on the dialog line.

Writing Extension Functions

This subsection describes features common to most extensions, and covers the following topics:

- Receiving Parameters
- Using PWB Functions to Get a File Handle
- Using PWB Functions to Read Lines From Files
- Using PWB Functions to Write Lines to Files
- Summary of PWB Functions

Because many PWB internal functions are public, your extension functions can also use them. Thus, this subsection demonstrates how to use the most common PWB functions--ones that manipulate the current file, and that list the callable PWB functions. However, for complete information on specific PWB functions, refer to online Help.

Receiving Parameters

Like native PWB functions, extension functions can receive parameters from users. For example, `Center.c` allows a user to select a range of lines to center with the `CenterLine` function.

Extension functions receive parameters similar to the way normal C programs receive command-line parameters. In both cases, parameters are passed in a predefined data construct. Normal C programs use **argc** and **argv**. Extension functions use the following parameters:

Parameter	Description
<code>argData</code>	The keystroke used to invoke your function
<code>pArg</code>	A pointer to a structure containing arguments passed to your function
<code>fMeta</code>	TRUE (nonzero) if meta precedes the argument, otherwise FALSE (zero)

The first parameter is rarely used. Most extension functions receive all their parameter data in the second parameter, `pArg`. This parameter is a pointer to a structure of type `ARG`, which contains:

Parameter	Description
<code>argType</code>	An integer that indicates the argument type
<code>arg</code>	A union of structures, one structure for each argument type

Typically, your function should test `pArg->argType` to find out which type of parameter PWB has passed. Once the type is known, the function responds accordingly. The following code from `Center.c` handles two argument types:

```
switch( pArg->argType )
    /* No argument.  Center current line */
    { case NOARG:
        yStart = yEnd = pArg->arg.noarg.y;
        break;
      /* Center range of lines */
      case LINEARG:
        yStart = pArg->arg.linearg.yStart;
        yEnd = pArg->arg.linearg.yEnd;
        break;
    }
```

If your function takes one argument only, it does not need to test `pArg->argType`. PWB knows beforehand which argument types your function accepts (via `cmdDesc`) and rejects invalid arguments.

Once the argument type is known, your function can access the parameters through `pArg->arg`, a structure whose members differ for each argument type. In the NOARG (no arguments) case, it contains `x` and `y` values identifying the cursor position in the current file:

```
struct noargType
{
    /* no argument */
    LINE y; /* cursor line */
    COL x; /* cursor column */
};
```

The `Center.c` example uses the `y` value in this structure (`noarg.y`, the cursor line) to center the current line:

```
    /* No argument.  Center current line */
case NOARG:
    yStart = yEnd = pArg->arg.noarg.y;
    break;
```

Similarly, in the LINEARG case, the pArg-->arg structure contains three values:

```
struct lineargType
{
    int    cArg;        /* line argument specified */
    LINE yStart;        /* count of args pressed */
    LINE yEnd;          /* starting line of range */
    LINE yEnd;          /* ending line of range */
};
```

The Center.c example uses the starting and ending values in this structure (yStart and yEnd) to center a range of selected lines:

```
case LINEARG: /* Center range of lines */
    yStart = pArg->arg.linearg.yStart;
    yEnd = pArg->arg.linearg.yEnd;
    break;
```

The method is the same for other argument types. For details about pArg-->arg structures for all argument types, refer to online Help.

Using PWB Functions to Get a File Handle

While extension functions can do many tasks, they typically manipulate a file in some way. For example, the CenterLine function in Center.c rewrites a line or lines in the current file. The current file appears in the editing window. Since it is open for editing, you can access the current file without opening it by assigning its file handle to a variable in your function.

PWB file-handling functions use file handles of type PFILE. The Center.c example declares the following handle variable:

```
PFILE pFile;
```

The FileNameToHandle function gets a handle to a file that is already open for editing:

```
pFile = FileNameToHandle( "", "" );
```

The function takes two string arguments. If the first string is null as shown here, the FileNameToHandle function returns a handle to the current file. To get handles to other files, use the AddFile function (in which case you may need to use other PWB functions such as FileRead).

Reading a Line from the File

To read a file once your function has a file handle, you can use the `GetLine` function, which reads one line at a time:

```
len = GetLine( yStart, buf, pFile );
```

The first argument is a line number, the second a pointer to a buffer, and the third a file handle. So this call reads line number `yStart` from the file whose handle is `pFile` into the buffer `buf`. Note that the first line in a file is line 0, not line 1.

Once you read a line into a local buffer, you can manipulate it as desired. `Center.c` uses the buffer `buf` to center the line's text.

Writing a Line to the File

To write lines back to files after modification, you can use the `PutLine` function, which writes one line at a time:

```
PutLine( yStart, buf, pFile );
```

`PutLine` takes the same arguments as `GetLine`: line number, buffer pointer, and file handle. In `Center.c`, this call writes the line from `buf` to line `yStart` in the file whose handle is `pFile`.

Summary of PWB Functions

If you understand how `Center.c` works, you understand the basics of how to use PWB functions in your own functions. The rest is a matter of learning details about individual functions. Table 12-1 lists the PWB functions, grouping them by category. For details about specific functions, refer to online Help.

Table 12-1. Callable PWB Functions

Category	Function	Description
Block Operations	CopyBox	Insert rectangular area
	CopyLine	Insert range of lines
	CopyStream	Insert stream of text
	DelBox	Delete rectangular area
	DelLine	Delete range of lines
	DelStream	Delete stream of text
Build	fGetMake	Get extmake setting
	SetMake	Set extmake setting
Cursor	GetCursor	Get cursor position
	MoveCur	Move cursor
Dialog	DoMessageBox	Create message dialog
	PopUpBox	Display text in dialog window
Display	BadArg	Report that argument was invalid
	Display	Update screen
	DoMessage	Display message on dialog line
File	AddFile	Open new file and get file handle
	DelFile	Delete contents of file buffer

continued

Table 12-1. Callable PWB Functions (cont.)

Category	Function	Description
	fChangeFile	Change current file to named file
	FileNameToHandle	Get handle to open file
	FileRead	Copy disk file to file buffer
	FileWrite	Copy file buffer to disk file
	pFileToTop	Make specified file the current file
	RemoveFile	Remove file from memory
Keyboard	KbHook	Restore keyboard control to PWB
	KbUnHook	Remove keyboard control from PWB
	ReadChar	Get information on next keystroke
Format	ReadCmd	Get keystroke information in CmdDesc
Line	FileLength	Get length of file
	GetLine	Get line from file
	PutLine	Write line to file
List	GetListEntry	Get item from list
	ScanList	Process list
Memory	Falloc	Allocate _far memory

continued

Table 12-1. Callable PWB Functions (cont.)

Category	Function	Description
Miscellaneous	Fdalloc	Deallocate _far memory
	fExecute	Execute macro
	FindSwitch	Get information about switch
	GetEditorObject	Get internal PWB data item
	GetString	Get input from dialog line
	mgetenv	Get environment string
	NameToFunc	Get information about function or macro
	NameToKeys	Get key(s) assigned to specified function
	Replace	Replace character
Search	SetEditorObject	Set internal PWB data item
	SetKey	Assign function to keystroke
	REsearch	Search for regular expression
Virtual Memory	search	Search for string
	fpbtoVM	Copy data to virtual memory
	VMalloc	Allocate virtual memory
	VMFree	Free virtual memory

continued

Table 12-1. Callable PWB Functions (cont.)

Category	Function	Description
Window	VMtofpb	Copy data from virtual memory
	CloseWnd	Close window
	Resize	Resize window
	SplitWnd	Split window

Building, Using, and Debugging Extensions

Building and using extensions involves four steps:

1. Compiling
2. Linking
3. Loading
4. Assigning Keystrokes

You can debug your extensions with the CTOS Debugger, as explained at the end of this section.

Compiling Extensions

To compile extensions, the source (.c) file must include ext.h, the extension header file. Since an extension is not a standalone executable file, it does not have a main() function.

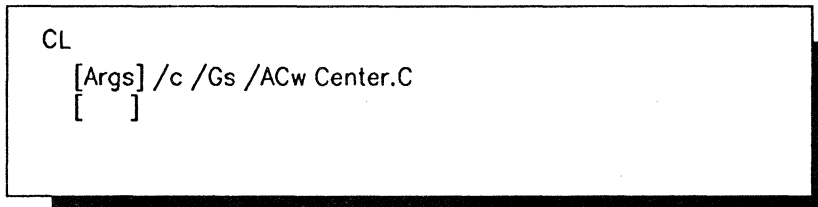
Figure 12-1 shows the easiest way to build extensions from the compiler command line. The Center.c extension appears for illustration. Notice that the MLINK switch /link /EXT appears at the end of [Args] line. This option automatically supplies the linker with the correct information for linking extensions.

```
CL
  [Args] /Gs /ACw Center.C /link /EXT
  [  ]
```

Figure 12-1. Sample CL Command Form for Building Extensions

The /Gs option turns off stack checking; the /ACw option selects the required custom memory model.

To disable automatic linking with MLINK and use MCBIND, compile with option /c and leave off the /link /EXT option as shown in Figure 12-2. Switch /c defeats automatic linking with MLINK. For additional details on the relationship between CL and MLINK, refer to Section 5, Building Applications.



```
CL
  [Args] /c /Gs /ACw Center.C
  [ ]
```

Figure 12-2. Sample CL Command Form for Compiling Extensions

Linking Extensions

To link extension object modules with MCBIND, you need to specify the startup module MC0CXT.Obj, the flavor module EXTHDRP.Obj, no stack, and the correct libraries. Other parameters for linking remain the same as for linking normal C programs, such as specifying protected mode and no DS allocation.

Figure 12-3 shows how to complete the MCBIND command form that comes with CTOS Microsoft C.

MCBind	
Object modules	MC0CXT.Obj EXTHDRP.Obj Center.Obj
Run file	Center.Cxt
[List file]	
[Publics?]	
[Line numbers?]	
[Stack, Heap size]	0
[Max array, data]	0 0
[Min array, data]	0 0
[Runfile mode]	Protected
[Version]	
[Libraries]	CLIBCP.LIB EXTSUP.LIB
[DS allocation?]	No
[Symbol file]	
[Copyright notice?]	
[File to append]	
[Linker configfile]	
[Source debugger]	

Figure 12-3. Sample MCBIND Command Form for Building Extensions

Loading Extensions

You can load extensions two ways:

- Automatically: Add **pwb** to the extension name
- Manually: Prepend **load:extension.cxt** to TOOLS.INI

To load extensions automatically, simply prepend **pwb** to the extension name. Upon startup, PWB loads all extensions prefaced with PWB.

For example, to load Center.Cxt automatically, rename it to **PwbCenter.Cxt**. Upon startup, PWB loads the extension.

To load extensions manually in TOOLS.INI, open the file, find [PWB], and add this line:

load:extension.cxt

For example, to load Center.Cxt, verify that your working directory contains the file, open TOOLS.INI with **Edit**, find [PWB], and add **load:center.cxt**. The entry should appear like this:

```
[PWB]
load:center.cxt
```

Upon startup, PWB loads the extension.

For both methods, PWB also adds the extension to the **Options, Editor Settings** file. For example, PWB adds these lines for Center.Cxt:

```
[pwb-center]
;   Numeric Switches

;   Boolean Switches

;   Text Switches
```

Assigning Keystrokes to Extension Functions

Before you can use extension functions in PWB, you must assign keystrokes to the functions. When PWB starts and loads extensions, it creates a section and assignment lines for each function in **Options, Key Assignments**, using this form:

```
[pwb-extension]

;      function:is unassigned
```

For example, if you load Center.Cxt from the TOOLS.INI file, PWB adds these lines to **Options, Key Assignments**:

```
[pwb-center]

;      CenterLine:is unassigned
```

If you load Center.Cxt automatically by renaming it to PwbCenter.Cxt, PWB adds these lines to the **Options, Key Assignments** file:

```
[pwb-pwbcenter]

;      CenterLine:is unassigned
```

Notice that PWB automatically detected the CenterLine function created with the Center.Cxt extension, and prepared it for key assignment.

As a result, you can assign keystrokes to functions in two ways:

- From PWB, edit **Options, Key Assignments**
- From the command line, **Edit TOOLS.INI**

To assign keystrokes from PWB, use this procedure:

1. Select **Options, Key Assignments**, and find **[pwb-extension]**.
For example, find **[pwb-center]**.
2. For each function that appears, delete the semicolon and the statement is **unassigned**.
3. For each function, add assignable keystrokes after the colons.

4. Click the mouse on a different line.

If the keystroke is valid, PWB highlights the assignment. If the keystroke is invalid, PWB displays an error message .

5. Choose one option:

- To assign the keystroke for the current session only, select **File, Close**.
- To assign the keystroke for all sessions, select **File, Save, Close**.

For example, to assign the keystroke SHIFT-F5 to CenterLine for all sessions, create and **Save** the following entry:

```
[pwb-center]
CenterLine:Shift+F5
```

If you **Save** for all sessions, PWB also reproduces the above entry in **TOOLS.INI**.

To assign keystrokes from the command line, use this procedure:

1. Open **TOOLS.INI** with the **Edit** command.
2. Add these lines:

```
[pwb-extension]
function:keystrokes
```

For example, the following lines assign SHIFT-F5 to the CenterLine function for **CENTER.EXT**:

```
[pwb-center]
CenterLine:Shift+F5
```

3. Exit the file.

Upon startup, PWB also reproduces the above entry in the **Options, Key Assignments** file.

As you can see, using either method assigns keystrokes to functions in two places. If you assign keystrokes in PWB from **Options, Key Assignments**, the assignment also appears in TOOLS.INI. If you assign keystrokes in TOOLS.INI, the assignment also appears in **Options, Key Assignments**. Thus, choosing a method defaults to personal preference only.

To invoke the function CenterLine while in PWB, press **SHIFT-F5**.

Debugging Extensions

You cannot debug PWB extensions with CodeView, because extensions are loaded GDT-based into PWB's partition using CTOS's LoadRunFile call. CodeView cannot debug GDT-based run files. You must, therefore, use the CTOS Debugger to debug PWB extensions.

Further, since a PWB extension is not a normal CTOS process, customary debugging procedures for invoking the Debugger and loading the symbol file will not work. Thus, you must observe the following rules and procedures to use the Debugger for extensions:

- You can only debug extensions on CTOS II 3.3 or CTOS III.
All other OSs incorrectly align symbols to an extension's code and data.
- The extension should have one code segment and one data segment only.
If you have multiple code segments, use the CTOS Linker's "PackCode" config file option to collect multiple code segments into a single code segment.

- The DATA segment must precede the CODE segment. To achieve the required segment ordering, specify the following CTOS Linker :ClassOrder: config file option:

```
:ClassOrder: (BEGDATA DATA CONST MSG BSS ENDBSS CODE)
```

Note: *To automatically accomplish the above tasks, use **MLINK** and the **/EXT** option. This option also automatically links your extension as a compact model, protected mode run file, using the correct startup modules and libraries, with no stack.*

- After linking your extension, use **DEBUG FILE** on the Extension and change the first instruction, at mc0start, into a breakpoint:

```
mc0start (MARK) NOP
```

Type

```
debug (RETURN)
```

When the extension loads and this breakpoint instruction executes, the Debugger displays the CS:IP of the breakpoint:

```
Debugger call at CS:IP in process nn
```

The nn is not the process number of the extension, it's the process number of PWB. If you are running under CTOS III, just load the extension symbol file in the normal manner. If you are running under CTOS II 3.3 or greater, you must calculate and provide the correct offset.

To calculate the extension symbol file load offset, subtract 12 (0x0c) from the CS value in a CODE-F command:

```
CS-0C, 'extension.Sym' (CODE-F)
```

If the offset is correct, the Debugger will display the version string you specified when you linked the extension. To confirm the correct alignment of code symbols, display the code in one of our extension's functions.

The success of this technique depends on the extension being loaded into contiguous GDT selectors (i.e. code and data selectors differing numerically by eight). When CTOS is initially booted, the acquisition of two contiguous selectors for an extension load is, in our experience, a certainty.

Over time, however, the probability of the two selectors being contiguous decreases as selectors are acquired and released in support of CTOS system activity. If this symbol alignment procedure fails, re-boot CTOS and try again.

This Page is Blank
Do Not Print

Section 13

Debugging C Programs with CodeView

This section describes how to use the CTOS Microsoft CodeView debugger, and reveals techniques you can use to locate errors in your programs. With CodeView, you can control the flow of execution, and display and modify variables and memory.

This section covers these topics:

- Preparing a Debug Build from the Command Line
- Understanding the CodeView Debugger Windows
- Overview of Debugging Techniques
- Controlling Execution
- Viewing and Modifying Program Data
- Replaying a Debug Session
- Advanced CodeView Debugger Techniques
- Customizing the CodeView Debugger from TOOLS.INI

The CTOS Microsoft CodeView debugger supports the Unisys mouse. Thus, this section describes how to execute most operations first with the mouse, then with keystrokes. For a list of startup command options, refer to Appendix B.

Note: *If you are familiar with the DOS version of Microsoft C, note that for CTOS the ALT key maps to COPY, and CTRL maps to CODE. In addition, pseudofiles for the DOS version are enclosed with angle brackets < >. For CTOS the pseudofiles are enclosed in parentheses (. For example, <assign> maps to (assign).*

Preparing a Debug Build from the Command Line

Before CodeView can debug a file, you must prepare a debug build that creates a file with the extension `.Cdv`. To prepare a debug build, you must compile and link with specific options.

The following subsections describe how to create a debug build with Executive commands, and cover these topics:

- Compiling and Automatic Linking with `MLINK`
- Compiling and Linking with `MCBIND`
- Starting the CodeView Debugger

All command forms shown in these subsections illustrate how to prepare a source file named `dollar.c`, which calculates the number of ways to make change for a dollar.

```
/*    dollar.c    */
#include <stdio.h>

void main()
{
    int half, quarter, dime, nickel, penny, occurrences=0;

    for (half =0; half < 3; half ++)
        for (quarter=0; quarter< 5; quarter++)
            for (dime =0; dime < 11; dime ++)
                for (nickel =0; nickel < 21; nickel ++)
                    for (penny =0; penny <101; penny = penny+5)
                        if ((half*50 + quarter*25 + dime*10 + nickel*5 + penny)
                            ==100)

                            occurrences++;

    printf("%d ways exist to make change for a dollar.",
        occurrences);
}
```

Note: *For complete information on how to create a CodeView build from the Programmer's WorkBench (PWB) and start the CodeView debugger from PWB, refer to Section 3, Using the Programmer's WorkBench.*

Caution:

CodeView is not compatible with the CTOS Librarian because the Librarian modifies object modules during processing. Thus, modules compiled for CodeViewing and added to a CTOS library before linking cannot be CodeViewed at source level. MLIB does not modify object modules, so use MLIB to perform any and all library work.

Compiling and Automatic Linking with MLINK

To compile a file for the CodeView debugger and automatically link with MLINK, you must add CodeView information with option `/Zi` (which causes the linker to link for CodeView), and suppress optimization with option `/Od`. The example shown in Figure 13-1 illustrates how to properly compile and link `dollar.c` using the CL command.

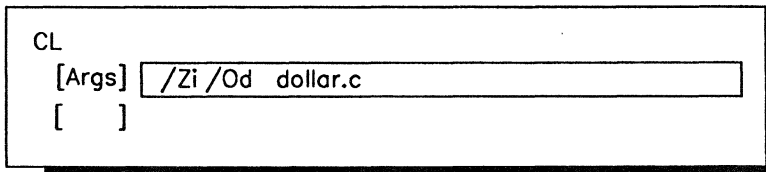


Figure 13-1. Sample CL Command with Linking for a CodeView Build

Note: This sample defaults to the small memory-model library.

Compiling and Linking with MCBIND

To compile a file for the CodeView debugger and link with MCBIND, you must disable automatic linking with option `/c`, and compile with options `/Zi` and `/Od`. Figure 13-2 illustrates how to complete the CL command form.

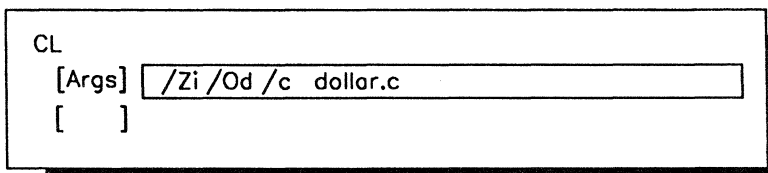


Figure 13-2. Sample CL Command without Linking for a CodeView Build

Once you compile your file, complete the MCBIND command form, and add the word **CodeView** to the **[Source Debugger]** field. This option causes the linker to create the file *your_program.Cdv*. For example, the sample link shown in Figure 13-3 creates a file named *dollar.Cdv*.

```

MCBIND
Object modules      MCO.Obj Dollar.Obj
Run file            Dollar.Run
[List file]
[Publics?]
[Line numbers?]
[Stack, Heap size]  8192
[Max array, data]   0 0
[Min array, data]   0 0
[Runfile mode]      protected
[Version]
[Libraries]          SLIBCP.LIB LIBCEXT.LIB
[DS allocation?]     no
[Symbol file]
[Copyright notice?]
[File to append]
[Linker configfile]
[Source debugger]    CodeView
    
```

Figure 13-3. Sample MCBIND Command for CodeView Build

Note: *This sample specifies the small memory-model library with **SLIBCP.LIB**.*

Starting the CodeView Debugger

For CodeView to work properly, verify that the following conditions exist:

- Task Management Service is installed.
- All files associated with the debugee are in your working directory.
- TOOLS.INI exists and is locatable by CodeView.

For example, to debug dollar.run:

- You must install Task Management Service either with a command or from SysInit.Jcl.
- Dollar.c, dollar.run, and dollar.Cdv must be in your working directory.
- Either TOOLS.INI must be in your working directory, or Environment Service must specify its path with INIT.

Note: *The CTOS Microsoft CodeView debugger works without locating TOOLS.INI, but displays a warning message in the Command window.*

The procedure that follows does not describe how to start the CodeView debugger from PWB. For details, refer to Section 3, Using the Programmer's WorkBench. (PWB).

To prepare your system, start CodeView, and display a file to debug, use the following procedure:

1. If you have not installed Task Management Service either with a command or from SysInit.Jcl, issue the command Install Task Management Service.
2. Verify that your working directory contains the debuggee, related debuggee files, and TOOLS.INI.
3. Path to your working directory.
4. Issue one of the following two commands:

```
CV
[Args] /options your_runfile
[      ]

CV Run
[[options] runfile]  your_runfile
[Case]
[Command]
[Parameter 1]
[Parameter ...]
[Parameter 16]
```

CodeView appears, and displays the source file in the center window.

For example, the following command invokes CodeView with the sample file dollar.run:

```
CV
[Args] dollar
[      ]
```

Notes: The .RUN suffix is not necessary. For a list of startup options and commands, refer to Appendix B, Command Reference.

The command CV RUN works like the Executive RUN command. If your programs use Executive command forms, and require or allow user input in parameter fields, you debug your program in CodeView with parameters by entering them in the [Parameter . . .] fields.

Understanding the CodeView Debugger Windows

So that large amounts of information can appear in an organized and easy-to-read fashion, CodeView divides the screen into windows. Each window operates independently, and has a distinct function. Each window name appears on top of the window, as listed in Table 13-1.

Table 13-1. List of CodeView Windows

Window Name	Description
Source	Displays source code. You can open a second Source window to view an include file, another source file, or the same source file at a different location.
Command	Accepts debugging commands.
Watch	Displays current values of selected variables.
Local	Lists values of all variables local to the current function or block.
Memory	Shows memory contents. You can open a second Memory window to view a different section of memory.
Register	Displays contents of the microprocessor's registers as well as the processor flags.
8087	Displays registers of the coprocessor or its software emulator

When first invoked for a file, CodeView appears with three windows. As shown in Figure 13-4, the Local window appears on top, the Source window fills the middle of the screen, and the Command window appears on the bottom.

To alter how the source appears, press F3. The Source window toggles between Assembly, C, or a mixture of the two languages, depending on the options specified under **Options, Source Window Options**.

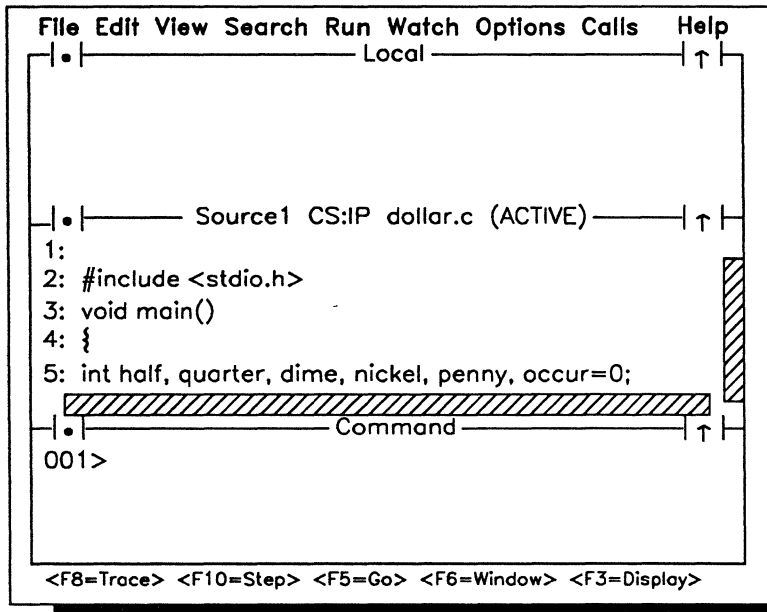


Figure 13-4. Sample CodeView Startup Screen

With subsequent startups, CodeView displays the screens and information that appeared when you exited. CodeView keeps this information in the file `CURRENT.STS`. CodeView creates this file either in the directory pointed to by the `INIT` environment variable (if it exists), or in your working directory. To cause CodeView to return to the initial three-screen startup, delete or edit `CURRENT.STS` (with due caution).

Windows open in two ways: manually and automatically. Manually, you can open windows from the **View** menu. You can also open multiple sessions for two windows, Source and Memory. Automatically, some windows open if you do certain operations. For example, if you select a Watch variable, the Watch window appears if not already open.

CodeView continually and automatically updates the contents of all windows, open or not. However, to interact with a particular window --set breakpoints, modify variables, and so forth--you must select the window to make it active.

The CodeView debugger identifies the active window three ways:

- Highlights the window's name in white.
- Moves the cursor to the window.
- Moves the vertical and horizontal scroll bars to the window.

To select a new active window, click left in the window (position the mouse cursor in the window and press the left mouse button). To toggle the active window around the screen with keystrokes, press F6 or SHIFT-F6.

Windows often contain more information than can appear in the area allocated to the window. To display additional information, either enlarge the window or scroll text. To enlarge the window, drag the mouse on the window's horizontal or vertical scroll bars (click and hold the left button; then drag). To scroll text, either click the mouse on the scroll bars, or press the arrow keys (LEFT, RIGHT, UP, DOWN).

Typing commands into the Source window causes CodeView to temporarily shift focus to the Command window. CodeView appends what you type to the last line in the Command window. If the Command window is closed, CodeView beeps and ignores the input. For a list of CodeView commands, refer to Appendix B.

Adjusting Windows

Although you cannot change window positions, you can size and close them. To make these adjustments, select **Maximize**, **Size**, and **Close** from the View menu; press CODE-F10, CODE-F8, and CODE-F4 respectively; or use the mouse:

- To maximize a window (fill the screen), at the right end of the window's top border, click left on the up arrow (which becomes a double arrow). To restore the window to its previous size and position, click left on the double arrow.
- To size a window, position the mouse pointer anywhere along the white line at the top of the window. Press and hold the left mouse button. When two double arrows appear on the line, drag the mouse to enlarge or reduce the window. The same action on a vertical border widens or narrows the window.
- To close a window, at the left end of the top border, click left on the dot. You can also close a window from the View menu whose name has a dot next to it by clicking on the name, or by pressing the window's accelerator key. The adjacent windows expand automatically to recover the empty space.

Overview of Debugging Techniques

Because no single approach to debugging is best for all programs and users, CodeView offers a variety of debugging tools to let you pick a method appropriate to your program and work habits. The subsections that follow may help you decide how to approach debugging, and cover these topics:

- Controlling Execution
- Viewing and Modifying Program Data
- Advanced CodeView Technique

Broadly speaking, two things can fail in a program:

- Execution flow
- Data Manipulation

Occasionally, these problems overlap. Incorrect execution can corrupt data, and bad data can cause incorrect execution. With CodeView, however, you do not have to know beforehand if the problem is bad flow, data manipulation, or a combination of both.

To monitor flow and variables simultaneously, CodeView features let you:

- Monitor and pause execution flow.
- View and modify any program variable, any section of memory, or any processor register.

Controlling Execution

CodeView allows two forms of program execution:

Continuous	The program executes until it either reaches a breakpoint, or terminates normally.
Single Step	The program pauses after executing each line of code.

The following subsections explain how each form works, and describe effective ways to use them.

Continuous Execution

Continuous execution lets you quickly execute the bug-free sections of code that would otherwise take a long time to execute by single stepping.

The simplest way to execute continuously is to click right (button) anywhere on a line of code in the Source window (position the mouse pointer and press the right mouse button). The program executes at full speed to the beginning of this line, then pauses. To accomplish the same task with keystrokes, place the cursor on the line and press F7. To finish executing your program, continue clicking right on successive lines of code, or press F5. You cannot, however, continue execution by clicking on lines of code previously executed.

You can also pause execution at specific lines of code by setting breakpoints, which remain in your code until you disable or delete them. Setting breakpoints gives you greater control over continuous execution because you can attach conditions to breakpoints. For example, you can set a breakpoint that pauses execution when a variable reaches a specific value.

The following subsections cover these breakpoint topics:

- Selecting and Editing Breakpoint Lines
- Setting Breakpoint Values
- Using Breakpoints

Selecting and Editing Breakpoint Lines

You can skip parts of the program by setting one or more lines as breakpoints. The program executes at full speed up to the first breakpoint, then pauses. To continue program execution to the next breakpoint, press F5, or click left on the <F5=Go> icon at the bottom of the screen. To halt execution at any time, press CANCEL.

You can set as many breakpoints as you like; available memory is the only limitation. When you set a breakpoint, CodeView highlights the line. When you remove a breakpoint, the highlight disappears.

You can set and remove breakpoints several ways:

- Double-click left on the desired breakpoint line.
- Position the cursor on the desired breakpoint line, and press F9.
- Click on **Watch, Set Breakpoint**, and choose an option.

(You cannot remove breakpoints from this screen, shown in Figure 13-6, illustrated and described later in this subsection under Setting Breakpoint Values.)

A breakpoint line must represent executable code. You cannot select a blank, comment, or declaration line as a breakpoint (such as a variable declaration or preprocessor statement).

You can also set breakpoints at functions and addresses. To set a breakpoint at a function, enter its name in Location [] field in the Set Breakpoint dialog box as shown in Figure 7-5. To set a breakpoint at an address, enter the address in CS:IP form in the same field.

To edit breakpoints, click on **Watch, Edit Breakpoints**. The Edit Breakpoints dialog box appears, a sample of which is shown in Figure 13-5.

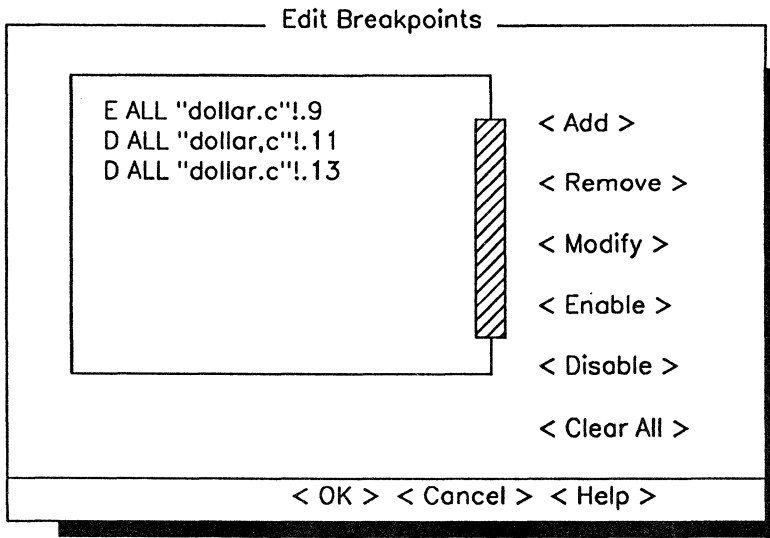


Figure 13-5. Edit Breakpoints Dialog Box

This dialog box shows all breakpoints. The three breakpoints listed are for source file `dollar.c`, and are located on lines 9, 11, and 13. The first letter, E or D, identifies whether the breakpoints are enabled or disabled, which you can control by clicking on a listed breakpoint, and then clicking on < Enable > or < Disable >. For example, to disable the first breakpoint, click on the first line, then click on < Disable >. E changes to D.

Note: *By default, the compiler optimizes your code. In the process of optimization, the compiler may reposition or reorganize some lines of code for more efficient execution. However, these changes can prevent CodeView from recognizing the corresponding lines of source code as breakpoints. Therefore, it is a good idea to disable optimization during development (use the /Od switch). You can restore optimization once debugging is completed.*

Setting Breakpoint Values

Breakpoints are not limited to specific lines of code. CodeView can also break execution when a variable changes or reaches a particular value. What is more, you can combine these value breakpoints with line breakpoints. With this technique, you can stop execution at specific lines when a variable changes or reaches a particular value.

To select these types of breakpoints, you must use the check boxes in the Set Breakpoint dialog box, shown in Figure 13-6.

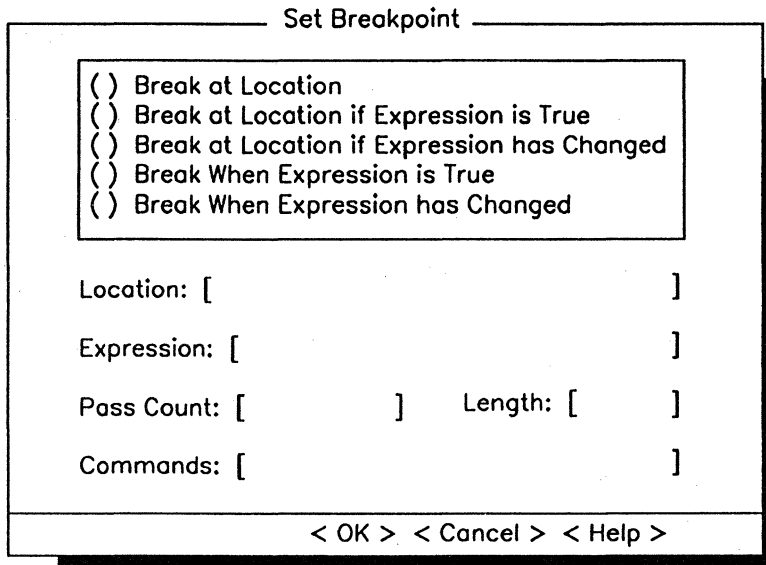


Figure 13-6. Set Breakpoint Dialog Box

To pause execution when an expression reaches a particular value, enter that expression in the Expression field. For example, assume you have declared a tree structure as follows:

```
struct Tagtree
{
    char * s;                /* Pointer to a string */
    struct TAGtree * left;   /* Pointer to left branch */
    struct TAGtree * right;  /* Pointer to right branch */
};

struct TAGtree t;
```

To pause execution when your tree traversal reaches a terminal node, enter the expression `(t.left == NULL) || (t.right == NULL)`.

To pause execution when a variable changes value, enter the variable name in the Expression field. To specify the number of bytes to check (for large variables such as arrays or character strings), enter the number of bytes in the Length field, up to a maximum of 32K.

Note: *When a breakpoint is tied to a variable, CodeView must check the variable's value after each machine instruction is executed. This slows execution greatly. For maximum speed when debugging, either tie conditional breakpoints to specific lines, or set conditional breakpoints only after you have reached the section of code that needs to be debugged.*

Using Breakpoints

The following examples show how breakpoints can help you find problems.

A common bug occurs when for loops execute too many or too few times. If you set a breakpoint that encloses loop statements, the program pauses after each iteration. By viewing loop or critical-program variables in the Watch or Local windows, you can see what the loop is doing wrong.

You do not have to pause at a breakpoint the first time execution reaches it. CodeView lets you specify the number of times to ignore the breakpoint condition before pausing. To specify the number of unpaused loops, click on **Watch, Set Breakpoint**. In the dialog box, enter the decimal number in the Pass Count [] field.

For example, suppose your program repeatedly calls a function to create a binary tree. You suspect that something goes wrong with the process about halfway through. You could mark the line that calls the function as the breakpoint, then specify how many times this line executes before pausing. Running the program creates a representative (but unfinished) tree structure that you can examine from the Watch window. You can then continue your analysis by single stepping.

Another common programming error is caused by erroneously assigning a value to a variable. To solve this problem, enter the variable in the Expression field of the Set Breakpoint dialog box. Execution breaks whenever this variable changes value.

Breakpoints are a convenient way to pause programs so you can assign new values to variables. For example, if a limit value is set by a variable, you can change the value to see if it affects program execution. Similarly, you can pass a variety of values to a switch statement to see if they are correctly processed.

This ability to alter variables is an especially convenient way to test new functions without having to write a standalone test program.

Single Stepping

When single stepping, CodeView pauses after executing each line of code. (If a line contains more than one executable statement, CodeView executes all the statements on the line before pausing.) The next executable line appears in reverse video.

To single-step through programs, use the **Step** and **Trace** keystroke commands. **Step** (F10) steps over function calls. CodeView executes all the code in the function, but to you, the function appears to execute as a single step. **Trace** (F8) traces through every step of all functions for which CodeView has symbolic information. CodeView executes each line of the function as a separate step. (CodeView has no symbolic information about run-time functions; therefore, they execute as single steps.) You can also alternate between **Trace** and **Step**.

To **Trace** continuously through a program, select **Run, Animate**, or press F8. To change animation speed, select **Options, Trace Speed**. To halt animation at any time, press any key.

Viewing and Modifying Program Data

The CodeView debugger offers a variety of ways to display program variables, processor registers, and memory. You can also modify any of these values as the program executes.

This subsection covers these topics:

- Displaying Variables in the Watch Window
- Displaying Expressions in the Watch Window
- Displaying Arrays and Structures
- Displaying Array Elements Dynamically
- Using Quick Watch
- Displaying Memory
- Displaying Processor Registers
- Modifying Values of Variables, Registers, and Memory

Displaying Variables in the Watch Window

To add a variable to the Watch window, position the cursor on the variable in the Source window. Then click on **Watch**, **Add Watch**; or press CODE-W.

A dialog box appears with the variable's name in the Expression field as shown in Figure 13-7.

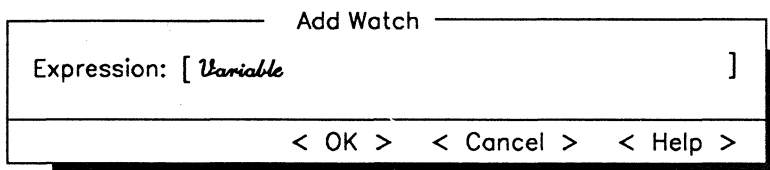


Figure 13-7. Add Watch Dialog Box

To change the variable, delete it, then type the name of the variable to watch. To add the variable to the Watch window, either click left on **OK** or press **GO**.

If the Watch window was not open, it appears at the top of the screen, and the variable appears in the window.

The following message may appear next to a newly-added variable:

<Watch Expression Not in Context>

This message appears when program execution has not yet reached the block where the variable is defined. (A block is a section of code enclosed in curly braces.) Global variables, however, (declared outside C functions) never cause CodeView to display this message; you can watch them from anywhere in the program.

To delete a variable from the Watch window, click on **Watch, Delete Watch**, and select the variable to be removed from the list in the dialog box. To delete the variable with keystrokes, position the cursor on any line in the Watch window, and press **CODE-Y**.

You can place as many variables as you like in the Watch window; available memory is the only limitation. CodeView automatically updates all watched variables as the program runs, including those not currently visible.

Because loops cause problems when they do not terminate correctly (do, for, or while), displaying loop variables in the Watch window is an easy way to determine whether a loop variable achieves its proper value.

Displaying Expressions in the Watch Window

As previously shown in Figure 13-7, the Add Watch dialog box specifically prompts for an expression, not a variable, although variables are valid entries. As this prompt suggests, you can enter an expression for CodeView to evaluate and display. An expression is any valid combination of variables, constants, and operators.

Sometimes expressions can be easier to interpret than components if you reduce several variables to a single, easily-read value. For example, imagine a for loop with two variables that have a constant ratio. However, you suspect that one of these variables sometimes takes the wrong value. By displaying (*var1* / *var2*) as an expression in the Watch window, you can see when values change without having to mentally divide two numbers.

You can also display Boolean expressions. For example, if a variable is never supposed to be larger than 100 or less than 25, the expression (*var* < 25 || *var* > 100) evaluates to 1 (true) when *var* goes out of bounds.

Displaying Arrays and Structures

Most program variables are scalar quantities--a single character or a single integer or floating-point value. These appear in the Watch window with the variable name to the left, followed by an equal sign (=) and the current value.

Arrays and structures contain multiple values arranged in one or more layers, and are often referred to as aggregate data items. CodeView lets you control how much of these variables to display; that is, all, part, or none of their internal structure.

An array initially appears in the Watch window in this form:

```
+wordholder[] = [...]
```

The brackets indicate that this variable contains more than one element. The plus sign (+) indicates that the variable is not yet expanded to display all components.

To expand the array, double-click anywhere on the line. You can also position the cursor on the line and press GO. For example, if *wordholder* is a six-character array containing the word "Basic," the Watch window display changes to the following:

```
-wordholder[]  
  [0] = 66  'B'  
  [1] = 97  'a'  
  [2] = 115 's'  
  [3] = 105 'i'  
  [4] = 99  'c'  
  [5] = 0   ''
```

Note that both the individual character values and their ASCII decimal equivalents appear. The minus sign (-) indicates that no further expansion is possible. To contract the array, double-click on its line again (or position the cursor on the line and press GO).

If it is inconvenient to view a character array in this form, cast the variable's name to a character pointer by placing (char *) in front of the name. The character array then appears as a string delimited by apostrophes.

You can display arrays with more than one dimension. For example, imagine a 5 x 5 integer array named matrix, whose diagonal elements are the numbers 1 through 5, and whose other elements are zero. Unexpanded, the array appears like this:

```
+matrix[] = [...]
```

Double-clicking on matrix (or pressing GO) changes the array to:

```
-matrix[]  
+ [0] [] = [...]  
+ [1] [] = [...]  
+ [2] [] = [...]  
+ [3] [] = [...]  
+ [4] [] = [...]
```

The actual values of the elements have not yet appeared. To view elements of the third row, position the cursor anywhere on the third row and press GO. The result looks like this:

```
-matrix[]  
+ [0] [] = [...]  
+ [1] [] = [...]  
- [2] []  
  [0] = 0  
  [1] = 0  
  [2] = 3  
  [3] = 0  
  [4] = 0  
+ [3] [] = [...]  
+ [4] [] = [...]
```

Expanding the fifth row produces this result:

```
-matrix[]
+ [0] [] = [...]
+ [1] [] = [...]
- [2] []
    [0] = 0
    [1] = 0
    [2] = 3
    [3] = 0
    [4] = 0
+ [3] [] = [...]
- [4] []
    [0] = 0
    [1] = 0
    [2] = 0
    [3] = 0
    [4] = 5
```

Any element of an array (or structure) can be independently expanded or contracted. To view one or two elements of a large array only, specify the particular array or structure elements in the Expression field of the Add Watch dialog box; you need not display every element of the variable.

You can dereference a pointer in the same way as you expand an array or structure. If you do, the pointer address appears, followed by all the elements of the variable to which the pointer currently refers. Also, multiple levels of indirection appear simultaneously (pointers referencing pointers).

Displaying Array Elements Dynamically

You do not have to display every element of an array. If you provide specific subscripts, the corresponding element appears only.

You can also specify dynamic array elements, which change as other variables change. For example, suppose that the loop variable *p* is a subscript for the array variable `catalogprice`. The Watch window expression `catalogprice[p]` displays the array element currently specified by *p* only, not the entire array.

Further, you can mix constant and variable subscripts. For example, the expression `bigarray[3][i]` displays only the element in the third row of the array to which the index variable *i* points.

Using Quick Watch

The Quick Watch window automatically expands arrays and structures to first levels. For example, an array with three dimensions is expanded to the first dimension.

Quick Watch allows you to quickly look at a variable or expression. However, since only one Quick Watch variable can be viewed at one time, you would not use Quick Watch to view most variables. Quick Watch works from three windows only: Source, Local, and Watch.

To select a variable to watch, position the cursor on the target variable, click left on **Watch**, **Quick Watch**, or press SHIFT-F9. The dialog box appears as shown in Figure 13-8.

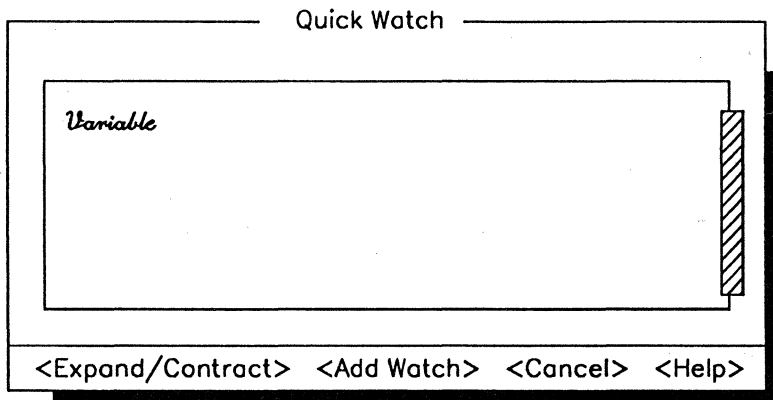


Figure 13-8. Quick Watch Dialog Box

If the variable is not the one you wanted, type in the desired expression or variable, then press GO. The selected item appears immediately.

You can also expand or contract an element just as you would in the Watch window: position the cursor on the appropriate line and press GO. To add a Quick Watch item to the Watch window, click left on **<Add Watch>**. Arrays and structures appear expanded in the Watch window as in the Quick Watch box.

Displaying Memory

To open a **Memory** window, select **View, Memory**. You can open up to two **Memory** windows at one time.

By default, memory appears as hexadecimal byte values, with 16 bytes per line. At the end of each line, the same memory appears in ASCII form. Values that correspond to printable ASCII characters (decimal 32 through 127) appear as ASCII. Values outside this range appear as periods.

Byte values are not always the most convenient way to view memory. If the memory area to examine contains character strings or floating-point values, you might prefer to view them in a readable form. To change the form, click on **Options, Memory Window**. The **Memory Window Options** dialog box appears, as shown in Figure 13-9.

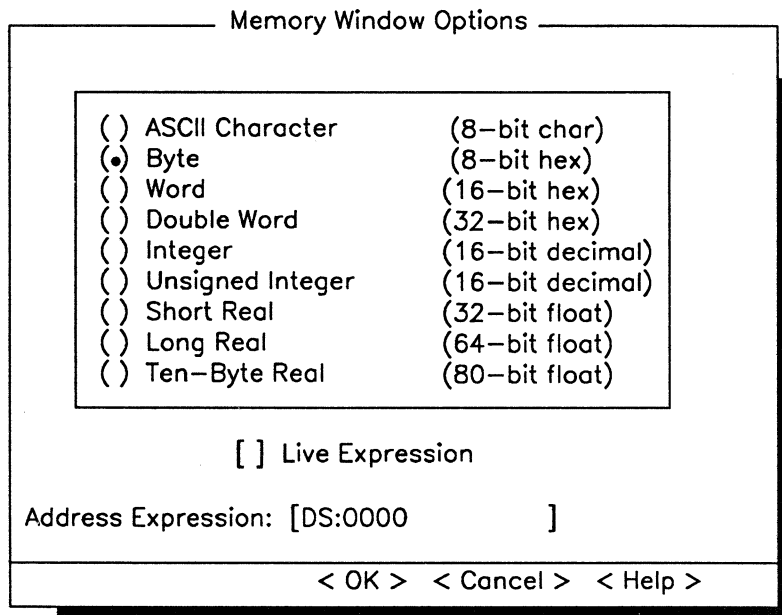


Figure 13-9. Memory Window Options

To select a different option, either click on the desired option, or press F3 until the desired option is selected.

If a section of memory cannot appear as a valid floating-point number, the number shown includes the characters NAN (not a number).

Displaying Variables with a Live Expression

From the Memory window, you can watch a particular memory area that your program uses to store data. This CodeView display feature is called a Live Expression.

Live means that the displayed memory area changes to reflect the value of a pointer or subscript. For example, suppose `buffer` is an array and `pbuf` is a pointer to that array, then `*pbuf` points to the array element currently referenced. A live expression displays the section of memory beginning with this element. If your program changes the value of `pbuf`, CodeView dynamically adjusts the Memory window display.

To create a live expression, click on **Options, Memory Window**. The Memory Window Options dialog box appears as shown previously in Figure 13-9. Click on the **Live Expression** box, and type the name of the element to view. For example, if `strgptr` is a pointer to an array of characters, and you want to see where it currently points, enter `*strgptr`. Then either click on **OK** or press **GO**.

A new Memory window opens. The first memory location in the window is the first memory location of the live expression. The section of displayed memory changes to the section that the pointer currently references.

Generally, it is more convenient to view an item in the Watch window than to view it as a live expression. However, some items are more easily viewed as live expressions. For example, to examine what is currently on top of the stack, enter `SS:SP` as the live expression.

Displaying Processor Registers

To view a processor registers, click on **View, Register**, or press F2. A window on the right side of the screen appears and shows the current values of the microprocessor's registers.

At the bottom of the window, a group of mnemonics appears, representing the processor flags. When you first open the Register window, all values appear in normal-intensity video. Subsequent changes appear in high-intensity video. For example, suppose the overflow flag is not set when the Register window is first opened. The corresponding mnemonic is NV and appears in light gray. If the overflow flag is subsequently set, the mnemonic changes to OV and appears in bright white.

You can also display the registers of an 8087/287/387 coprocessor in a separate window by selecting **View, 8087**. If your program uses the coprocessor emulator, the emulated registers appear instead.

Modifying Values of Variables, Registers, and Memory

You can easily change variable values, memory locations, or registers displayed in the Watch, Local, Memory, Register, and 8087 windows. To change values, position the cursor on the desired value and change it. To undo the last change, press COPY-BKSP.

Caution:

The effect of changing a register, flag, or memory location may vary from no effect at all, to crashing the operating system. You should be cautious when altering machine-level values; most of the items you would want to change can be altered from the Watch window.

The following subsections briefly mention features of two windows: Memory and Register.

Memory Window

At the left of the Memory window, the starting address of each line of memory appears in CS:IP form. When you alter addresses, the display shifts automatically to the corresponding section of memory. If your program cannot access that section, memory locations appear as double question marks (??).

When you select Byte display from the Memory Window Options dialog box, CodeView presents the data in both hexadecimal and ASCII. (Byte display is the default.) You can change data in memory either by entering new hex values over the hexadecimal representation of your data or by entering character values over the character representation.

Register Window

To toggle a processor flag, click left on its mnemonic. You can also position the cursor on a mnemonic, then press any key (except TAB or SPACE). Repeat to restore the flag to its previous setting.

One instance where direct manipulation of register values can be valuable is when you are debugging in-line assembly code. You can change register values to test assumptions before making changes in your source code and recompiling.

Replaying a Debug Session

CodeView can automatically create a tape (disk file) with all the debugging instructions and input data that you entered when testing a program. When replayed, the tape repeats the debugging process. This dynamic replay feature is unique to the CodeView debugger.

You can use the recording as a bookmark, so when quitting a long debugging session, you can return later to the same place. Principally, this feature allows you to back up when you make an error or overshoot the section of code with the bug. This feature is important because not all bugs are located when executing the program in a linear fashion.

For example, you may have to manually execute a function many times before its bug appears. However, if you lose the information that caused the bug by entering a command that alters the machine's or program's status, you will have to manually repeat every debugging step to return where the bug appeared. Even worse, if you do not remember the exact sequence of events that exposed the bug, it could take hours to recreate.

To record a tape, select **Run, History On**. To end a tape, select **History On** again.

To replay the tape, select **Run, Undo**. The program restarts automatically, and rapidly executes every debug command up to, but not including, the last one entered. You can repeat this process as many times as you like until you return to the desired point in execution.

To add additional steps to an existing tape, select **Run, History On**, then select **Replay**. When replay has completed, perform whatever new debugging steps you want. To end the recording, select **History On** a second time. The new tape contains both the original and added commands.

Note: CodeView records mouse commands that apply to CodeView only.

Advanced CodeView Debugger Techniques

Once you are comfortable stepping through programs, displaying and changing variables, and using dynamic replay, you might want to experiment with the following advanced techniques:

- Setting Command-Line Arguments
- Multiple Source Windows
- Calling Functions
- Checking for Undefined Pointers
- Printing Selected Items
- Handling Register Variables
- Redirecting CodeView Input and Output

Setting Command-Line Arguments

If your program retrieves command-line arguments, you can specify them by selecting **Run, Set Runtime Arguments**. Enter the arguments in the Command Line field before you begin execution. (Arguments entered after execution begins cause an automatic restart.)

Multiple Source Windows

You can open two Source windows at the same time. The windows can display two different sections of the same program, or one can show the high-level listing and the other the assembly-language listing. In the latter case, the contents of the windows track, with the next assembly-language instruction to be executed matching the next line of source code.

You can move freely between these windows, executing a single line of source code or a single assembly instruction at a time. The assembly-language window must be opened in CS:IP mode.

Calling Functions

You can call any C function in your program from the Command or Watch windows (whether user-written or from the library), using the following format:

```
?funcname (varlist)
```

The function is evaluated and the returned value appears in the Command window.

The function does not have to be called by your program to be available for evaluation. For example, all the .OBJ code specified in the linker input response file is linked. The functions in this code can then be evaluated from the Command window. This feature allows you to run functions from within CodeView that you would not normally include in the final version of your program.

Checking for Undefined Pointers

Until a pointer has been explicitly assigned a value, its value is undefined. That is, its value may be completely random, or it may be some consistent value that does not point to a useful data address (such as -1).

Accessing data through an uninitialized pointer will cause unpredictable program behavior, results in a protection violation. Because many C programs use pointers heavily, tracking down exactly which pointer variable was left uninitialized is tedious.

CodeView can help locate the problem quickly. If you use an uninitialized pointer the operating system generates a protection violation. By examining the Calls menu, you can determine the last line of your code that was executed before the protection violation occurred.

Printing Selected Items

You can print all or part of the contents of any window with the Print command from the File menu. The check box lets you print the complete contents of the window, the material currently viewable in the window only, or text selected from the window. To select text, either drag the mouse across it, or press **SHIFT-arrow key**.

By default, print output goes to the file CODEVIEW.LST in the current directory. To choose between appending or overwriting, select **File, Print**. The Print dialog box appears as shown in Figure 13-10.

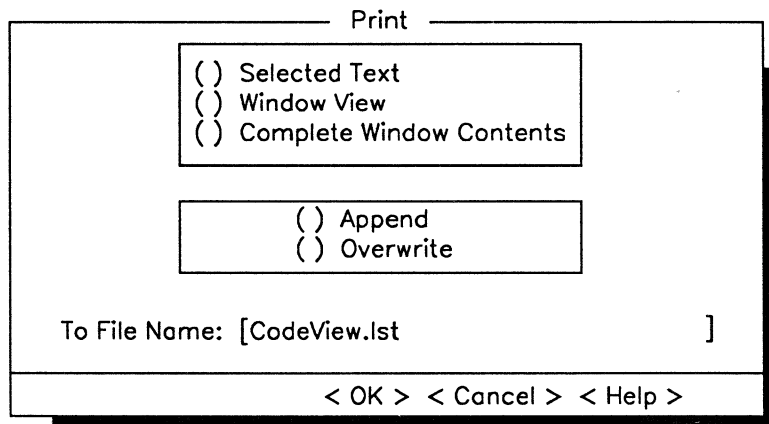


Figure 13-10. Print Dialog Box

From here, you can choose whether to overwrite or append to the file. To send output to a different file, change the file name in the To File Name field. To send the output to a printer, type the printer name in square brackets. For example, to send output to printer Spike, enter [Spike].

Handling Register Variables

Register variables are stored in microprocessor registers rather than RAM, which speeds access to variables. Conventional variables become register variables in two ways. First, you can declare them as register variables in source code; if registers are free, the compiler uses them. Second, if the compiler is optimizing and detects an often-used variable, such as a loop variable, it may convert the variable to a register variable.

However, register variables can cause problems during debugging. As with local variables, they are visible only within the function where they are defined. In addition, a register variable may not always appear with its current value.

Thus, in general it is good practice to avoid declaring register variables, and to turn off all optimization until you have fully debugged your program. You can then isolate any side effects produced by optimization or register variables.

Redirecting CodeView Input and Output

The Command window accepts DOS-like commands that redirect input and output. These commands can also be included on the command line that invokes CodeView. Whatever follows the /C option in the command line is treated as CodeView commands that are immediately executed at start-up.

```
CV/c "infile; t >outfile" myprog
```

Input is redirected to infile, which can contain start-up commands for CodeView. When CodeView exhausts all commands in the input file, focus automatically shifts to the command window. Output is sent to outfile and echoed to the Command window. The `t` must precede the `>` command for output to be sent to the Command window.

Redirection is a useful way to automate CodeView start-up. It also lets you keep a viewable record of command-line input and output, a feature not available with dynamic replay. (No record is kept of mouse operations.) Some applications, particularly interactive ones, may need modification to allow for redirection of input to the application itself.

Customizing the CodeView Debugger from TOOLS.INI

The TOOLS.INI file customizes the behavior and user interface of several CTOS Microsoft products. The TOOLS.INI file is a plain ASCII text file. You should place it in a directory pointed to by the INIT environment variable. (If you do not use the INIT environment variable, CodeView looks for TOOLS.INI only in its source directory.)

The CodeView section of TOOLS.INI is preceded by the following line:

```
[cv]
```

Most of the TOOLS.INI customizations control screen colors, but you can also specify such things as start-up commands or the name of the file that receives CodeView output. Online Help contains full information about all TOOLS.INI switches for CodeView.

This Page is Blank
Do Not Print

Section 14

Programming with Mixed Languages

Mixed-language programming allows you to make calls from programs written in one language to routines written in another. For example, you can code a CTOS Microsoft C program that calls programs written in other languages, and code programs in other languages that call C functions.

This section describes the elements of mixed-language programming, and covers the following topics:

- Making Mixed-Language Calls
- Library Conflicts
- Language Conventions
- Compiling and Linking
- C Calls to High-Level Languages
- C Calls to Pascal and PL/M
- C Calls to Assembly
- Handling Data

Making Mixed-Language Calls

Mixed-language programming always involves a call to a function, procedure, or subroutine. For example, a Pascal main module may need to execute a specific task that you would like to program separately. Instead of calling a Pascal subprogram, however, you decide to call a C function.

Mixed-language calls involve calling functions in separate modules. Instead of compiling all source modules with the same compiler, you use different compilers. In the instance mentioned above, you would compile the main-module source file with the Pascal compiler, another source file (written in C) with the C compiler, and then link the two object files.

Figure 14-1 illustrates how the syntax of a mixed-language call works, using the instance mentioned above.

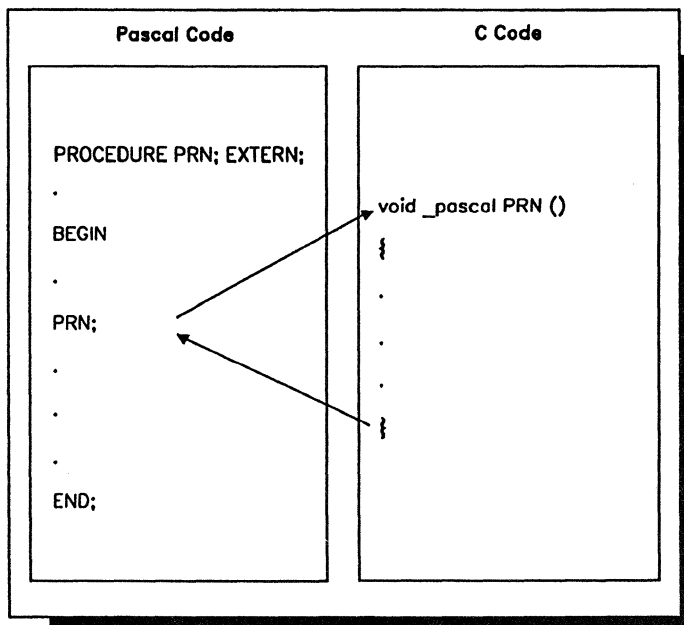


Figure 14-1. Mixed-Language Call

In Figure 14-1, the Pascal call to C is `PRN;`. There are two differences between this mixed-language call and a call between two Pascal modules:

- The subprogram `PRN` is implemented in C, using standard C syntax.
- The implementation of the call is affected by the C declaration statement, which uses the `_pascal` keyword to create compatibility with Pascal. The `PRN` prototype is an example of a mixed-language interface statement. These interface statements override default naming and calling conventions.

You can make mixed-language calls to routines regardless of whether they have return values. (In this section, routine refers to any function, procedure, or subroutine that can be called from another module.)

Table 14-1 shows the correspondence between calls to routines in different languages.

Table 14-1. Language Equivalents for Routine Calls

Language	Return Value	No Return Value
Assembly language	Procedure	Procedure
C	Function	(void) function
FORTRAN	FUNCTION	SUBROUTINE
Pascal	Function	Procedure

For example, a C module can make a subprogram call to a FORTRAN subroutine. You can prototype a FORTRAN subroutine as a function with a void type.

Library Conflicts

Unlike BTOS C, the Microsoft C Library design requires a DGroup organization that prohibits the DS Allocation for DGroup that Pascal requires. Because the two data segment architectures conflict, Pascal and C Library functions that provide resource management, such as memory management, file operations, floating point, and keyboard and video handling, cannot be combined in the same run file.

This means that a program with a main () function written in Pascal must have all of its resource-management code written in Pascal. Conversely, a program with a main () function written in CTOS Microsoft C must have all of its resource-management code written in CTOS Microsoft C.

Language Convention Requirements

To mix languages, the calling program must observe the same conventions as the called program. These conventions fall into the following categories:

- Naming Conventions
- Calling Conventions
- Parameter-Passing Conventions

Naming Convention Requirements

Both the calling program and the called subprogram must agree on the names of identifiers. Identifiers can refer to subprograms (functions, procedures, and subroutines) or to variables that have a public or global scope. Each language alters the names of identifiers.

The term naming convention refers to the way a compiler alters the name of the routine before placing it in an object file. Languages may alter the identifier names differently. You can choose between several naming conventions to ensure that the names in the calling program agree with those in the called program. If the names of called routines are stored differently in each object file, the linker will not be able to find a match. It will instead report unresolved external references.

CTOS compilers place machine code into object files; they also place the names of all publicly accessed routines and variables in object files. The linker can then compare the name of a routine called in one module with the name of a routine defined in another module, and recognize a match. Names are stored in the ASCII character set.

FORTTRAN and Pascal use similar naming conventions. They translate each letter to uppercase.

FORTTRAN and Pascal recognize the first eight characters of any name (unless identifier names are truncated). Names longer than eight characters get truncated.

The C compiler does not translate letters to uppercase. It inserts a leading underscore (`_`) in front of the name of each routine. C recognizes the first 31 characters of a name.

Differences in naming conventions are dealt with automatically by mixed-language keywords, as long as you follow two rules:

- FORTRAN routines compiled with the \$TRUNCATE metacommand enabled, make all names 6 characters or less. Thus, you must make all names 6 characters or less when using FORTRAN routines compiled with the FORTRAN compiler.
- When programming C modules, you must be careful not to rely upon differences between uppercase and lowercase letters.

Note: *When compiling with the command-line option /Gc (generate Pascal-style function calls), or declaring functions or variables with the `_plm` and `_pascal` keywords, the compiler does not insert a leading underscore and translates your identifiers to uppercase.*

Figure 14-2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

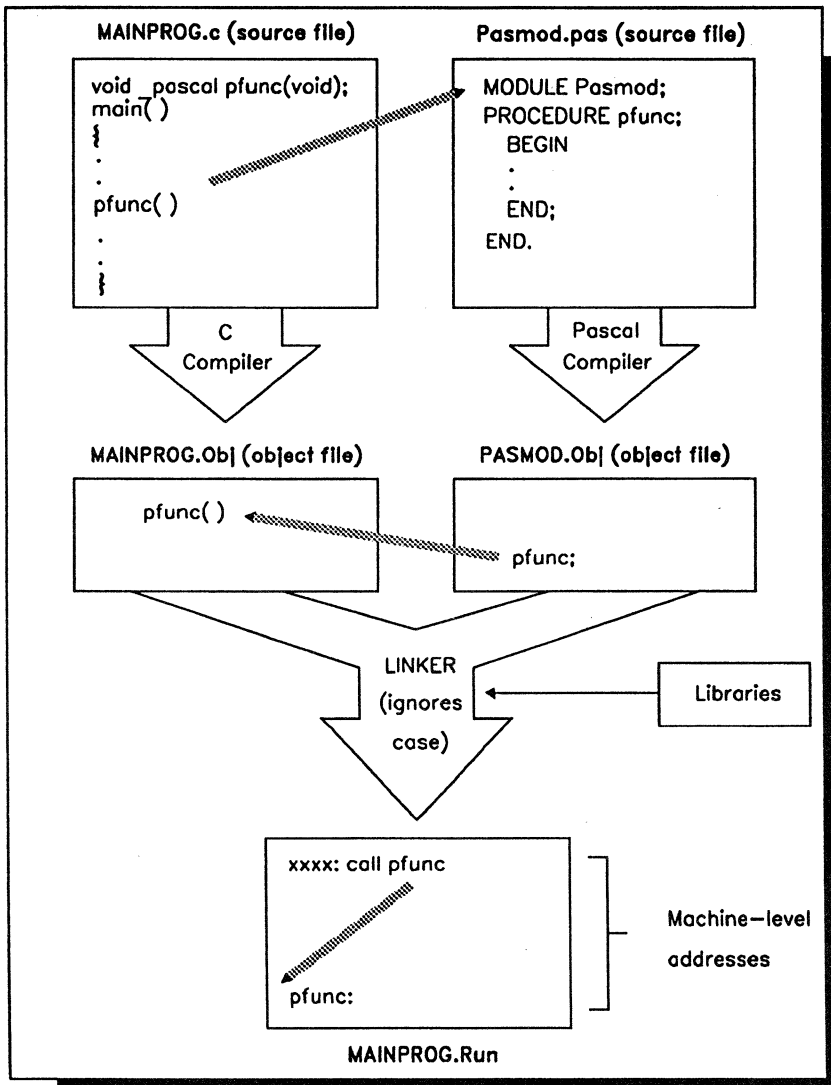


Figure 14-2. Naming Convention

Calling Convention Requirement

The term calling convention refers to the way a language implements a call. The choice of calling convention affects the machine instructions that a compiler generates to execute (and return from) a function, procedure, or subroutine call.

It is crucial that the two routines concerned (the routine issuing a call and the routine being called) use the same protocol. Otherwise, the processor may receive inconsistent instructions, causing the program to behave incorrectly.

The use of a calling convention affects programming in three ways:

- The calling routine uses a calling convention to determine the order in which to pass arguments (parameters) to another routine. This convention can be specified in a mixed-language interface statement or declaration.
- The called routine uses a calling convention to determine the order in which to receive the parameters passed to it. In C, this convention can be specified in the routine's heading.
- Both the calling routine and the called routine must agree on which of them is responsible for adjusting the stack after all parameters are removed.

In other words, each call to a routine uses a certain calling convention; each routine specifies or assumes some calling convention. The two conventions must be compatible. Usually, however, it is easier to adopt the convention of the called routine. For example, a C function would use its own convention to call another C function, and would use the Pascal convention to call Pascal.

FORTRAN and Pascal use the same standard calling convention. C uses a different convention.

Effects of Calling Conventions

Calling conventions dictate three things:

- The way parameters are communicated from one routine to another (in mixed-language programming, parameters or pointers to the parameters are passed on the stack)
- The order in which parameters are passed from one routine to another
- The part of the program responsible for adjusting the stack

The FORTRAN and Pascal calling conventions push parameters onto the stack in the order in which they appear in the source code. For example, the Pascal statement:

```
Calc(A, B);
```

pushes argument A onto the stack before it pushes B. These conventions also specify that the stack is adjusted by the called routine just before returning control to the caller.

The C calling convention pushes parameters onto the stack in the reverse order from their appearance in the source code. For example, the C function call:

```
calc(a, b);
```

pushes b onto the stack before it pushes a. In contrast with the other high-level languages, the C calling convention specifies that a calling routine always adjusts the stack immediately after the called routine returns control.

The FORTRAN and Pascal conventions produce slightly less object code. However, the C convention makes calling with a variable number of parameters possible. (Because the first parameter is always the last one pushed, it is always on the top of the stack; therefore it has the same address relative to the frame pointer, regardless of how many parameters were actually passed.)

Note: *The `_fastcall` keyword, which specifies that parameters are to be passed in registers, is incompatible with programs written in other languages. Avoid using `_fastcall` or the `/Gr` command-line option for C functions that you intend to make public to FORTRAN or Pascal programs.*

Parameter-Passing Requirements

Your programs must agree on the calling convention and the naming convention; they must also agree on the order in which they pass parameters. It is important that your routines send parameters in the same way to ensure proper data transmission and correct program results.

CTOS compilers support three methods for passing a parameter, as listed in Table 14-2.

Table 14-2. Methods for Passing Parameters

Method	Description
Near reference	Passes a variable's near (offset) address. This address is expressed as an offset from the default data segment. This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter changes the variable in the calling routine.
Far reference	Passes a variable's far (segmented) address. This method is similar to passing by near reference, except that a longer address is passed. This method is slower than passing by near reference, but is necessary when you pass data that is outside the default data segment. (This is an issue in Pascal only if you have specifically requested far memory.)
Value	Passes only the variable's value, not its address. With this method, the called routine knows the value of the parameter but has no access to the original variable. Changes to a value passed by a parameter have no affect on the value of the parameter in the calling routine.

These different parameter-passing methods mean that you must consider the following when programming with mixed languages:

- You must make sure that the called routine and the calling routine use the same method for passing each parameter (argument). In most cases, you will need to check the parameter-passing defaults used by each language and possibly make adjustments. CTOS Microsoft C has keywords or language features that allow you to change parameter-passing methods.
- You may want to choose a specific parameter-passing method rather than using the defaults of any language

Table 14-3 summarizes the parameter-passing defaults for each language.

Table 14-3. Parameter-Passing Defaults

Language	Near Reference	Far Reference	By Value
C	Near arrays	Far arrays	All data except arrays
FORTRAN	All (medium model)	---	---
Pascal	VAR CONST	VARS CONSTS	Other parameters

Compiling and Linking

After you have written your source files and decided on a naming convention, a calling convention, and a parameter-passing convention, you are ready to compile and link individual modules.

Compiling with Correct Memory Models

With FORTRAN and Pascal, no special options are available to compile source files that are part of a mixed-language program.

FORTRAN and Pascal use only far (segmented) code addresses. Therefore, you must use one of two techniques with C programs that call one of these languages:

- Compile C modules in medium, large, or huge model (using the /AX command-line options), because these models also use far code addresses
- Apply the `_far` keyword to the definitions of C functions you make public.

If you use the /AX command-line option to specify medium, large, or huge model, all your function calls become far by default. This means that you do not have to declare your functions explicitly with the `_far` keyword.

Choice of memory model affects the default data pointer size in C, although this default can be overridden with the `_near` and `_far` keywords. With C, choice of memory model also affects whether data objects are located in the default data segment; if a data object is not located in the default data segment, it cannot be passed by near reference.

C Calls to High-Level Languages

For calls to PL/M routines in the CTOS environment, follow the rules for calls to Pascal routines.

Just as you can call CTOS Microsoft C routines from other languages, you can call routines written in FORTRAN and Pascal from C. With FORTRAN, Pascal, and C, freestanding routines can be written with no restriction.

Executing Mixed-Language Calls

The C interface to other languages uses standard C prototypes, with the `_plm`, `_fortran` or `_pascal` keywords. Using any of these keywords causes the routine to be called with the PLM/Pascal naming and calling convention. Here are the recommended steps for executing a mixed-language call from C:

1. Write a prototype for each mixed-language routine called. The prototype should declare the routine extern for the purpose of program documentation.

Instead of using the `_fortran` or `_pascal` keyword, you can simply compile with the Pascal calling convention option `/Gc`. The `/Gc` option causes all functions in the module to use the PLM/Pascal naming and calling conventions, except where you apply the `_cdecl` keyword.

2. Pass the values of variables or pointers to variables. You can obtain a pointer to a variable with the address-of (`&`) operator.

In C, array names are always passed as pointers to the first element of the array; they are always passed by reference.

The prototype you declare for your function ensures that you are passing the correct length address (that is, near or far).

3. Issue a function call in your program as though you were calling a C function.
4. Always compile the C module in either medium, large, or huge model, or use the `_far` keyword in your function prototype. This ensures that a far (intersegment) call is made to the routine.

Using the `_plm`, `_fortran`, and `_pascal` Keywords

Two rules of syntax apply when you use the `_plm`, `_fortran`, and `_pascal` keywords:

- The `_plm`, `_fortran`, and `_pascal` keywords modify only the item immediately to their right.
- You can use `_near` and `_far` keywords with the `_plm`, `_fortran`, and `_pascal` keywords in prototypes. The sequences `_plm _far` and `_far _plm` are equivalent.

The keywords `_plm`, `_fortran`, and `_pascal` essentially have the same effect on the program; using one or the other makes no difference except for internal program documentation. Use `_plm` to declare a PL/M routine, `_fortran` to declare a FORTRAN routine, and `_pascal` to declare a Pascal routine.

The `_plm` keyword has been introduced to provide register protections for calls to CTOS.LIB and other PL/M functions. The `_plm` keyword applied to a function causes special code generation for each call to the function, which protects the SI and DI register contents, and clears the direction flag (DF). If you compile a `_plm` function, the compiler generates code as if the function was a `_pascal` function.

In addition, the `_fortran` keyword duplicates the effects of the `_pascal` keyword.

The following examples demonstrate the syntax rules:

1. This example declares `func` to be a PL/M, Pascal, or FORTRAN function taking two short parameters and returning a short value.

```
short _pascal func(short sarg1, short sarg2);
```

2. This example declares `func` to be pointer to a PL/M, Pascal, or FORTRAN routine that takes a long parameter and returns no value. The keyword `void` is appropriate when the called routine is a Pascal procedure or FORTRAN subroutine, since it indicates that the function returns no value.

```
void (_fortran * func)(long larg);
```

3. This example declares `func` to be a `_near` PL/M, Pascal, or FORTRAN routine. The routine receives a double parameter by reference (because it expects a pointer to a double) and returns a short value.

```
short _near _pascal func(_near double * darg);
```

4. This example is equivalent to the preceding example (`_pascal _near` is equivalent to `_near _pascal`).

```
short _pascal _near func(_near double * darg);
```

5. This example declares `func` to be a `_far` PL/M routine that takes an unsigned integer parameter and returns a short value.

```
short _plm _far func(unsignedf uarg);
```

C Calls to FORTRAN

Text that follows shows examples of C-FORTRAN programs that show the two types of subprogram calls to FORTRAN routines:

- Calls to Subroutines
- Calls to Functions

Functions return values; subroutines do not.

Calling a FORTRAN Subroutine from C

The example below demonstrates a C main module calling a FORTRAN subroutine, MAXPARAM. This subroutine adjusts the lower of two arguments to be equal to the higher argument.

```
/* C source file - calls FORTRAN subroutine
 * Compile in medium or large model */

extern void _fortran maxparam(int _near * I, int _near * J);

/* Declare as void, because there is no return value.
 * FORTRAN keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Two integer parameters, passed by near reference. */

main()
{
    int a = 5;
    int b = 7;
    printf("a = %d, b = %d", a, b);
    maxparam(&a, &b);
    printf("a = %d, b = %d", a, b);
}
```



```
C      FORTRAN source file, subroutine MAXPARAM
C
$NOTRUNCATE

      SUBROUTINE MAXPARAM (I, J)
      INTEGER*2 I [NEAR]
      INTEGER*2 J [NEAR]
C
C      I and J received by near reference,
C      because of NEAR attribute
C
      IF (I .GT. J) THEN
      J = I
      ELSE
      I = J
      ENDIF
      END
```

In the example, the C program adopts the naming convention and calling convention of the FORTRAN subroutine. The two programs must agree on whether parameters are to be passed by reference or by value. The following keywords affect how the two programs interface:

- The `_fortran` keyword directs C to call `maxparam` with the FORTRAN/Pascal naming convention (as `MAXPARAM`); `_fortran` also directs C to call `maxparam` with the FORTRAN/Pascal calling convention.
- Since the FORTRAN subroutine `MAXPARAM` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is specified in C by the use of near pointers, and in FORTRAN by applying the `NEAR` keyword to the parameter declarations.

Far reference could have been specified by using far pointers in C. In that case, you would not declare the FORTRAN subroutine `MAXPARAM` with the `NEAR` keyword. If you compile the FORTRAN program in medium model, declare `MAXPARAM` using the `FAR` keyword.

Calling a FORTRAN Function from C

The following example demonstrates a C main module calling the FORTRAN function fact. This function returns the factorial of an integer value.

```

/* C source file - calls FORTRAN function.
 * Compile in medium or large model. */

int _fortran fact(int N);

/* FORTRAN keyword causes C to use FORTRAN/Pascal calling and
 * naming conventions. Integer parameter passed by value. */

main()
{
    int x = 3;
    int y = 4;

    printf("The factorial of x    is %4d", fact(x));
    printf("The factorial of y    is %4d", fact(y));
    printf("The factorial of x+y is %4d", fact(x + y));
}

C      FORTRAN source file - factorial function
C
$NOTRUNCATE
      INTEGER*2 FUNCTION FACT (N)
      INTEGER*2 N [VALUE]

C
C      N is received by value, because of VALUE attribute
C
      INTEGER*2 I
      FACT = 1
      DO 100 I = 1, N
          FACT = FACT * I
100  CONTINUE
      RETURN
      END

```

In the example, the C program adopts the naming convention and calling convention of the FORTRAN subroutine. Both programs must agree on whether parameters are passed by reference or by value. Note that the C program passes the parameters by value rather than by reference. Passing parameters by value is the default for C.

To accept parameters passed by value, the keyword `VALUE` is used in the declaration of `N` in the `FORTTRAN` function. The `_fortran` keyword directs `C` to call `fact` with the `FORTTRAN/Pascal` naming convention (as `FACT`); `_fortran` also directs `C` to call `fact` with the `FORTTRAN/Pascal` calling convention.

When passing a parameter that should not be changed, pass the parameter by value. Passing by value is the default method in `C` and is specified in `FORTTRAN` by applying the `VALUE` attribute to the parameter declaration.

C Calls to Pascal and PL/M

Text that follows shows two examples of C-Pascal programs that show the two types of subprogram calls to Pascal routines:

- Calls to Procedures
- Calls to Functions

Functions return values; procedures do not.

Calling a Pascal or PL/M Procedure from C

The following example demonstrates a C main module calling a Pascal procedure, `maxparam`. This procedure adjusts the lower of two arguments to be equal to the higher argument.

```
/* C source file - calls Pascal procedure.
 * Compile in medium or large model. */

void _pascal maxparam(int _near * a, int _near * b);

/* Declare as void, because there is no return value.
 * The _pascal keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Two integer params, passed by near reference. */

main()
{
    int a = 5;
    int b = 7;

    printf("a = %d, b = %d", a, b);
    maxparam(&a, &b);
    printf("a = %d, b = %d", a, b);
}

{ Pascal source code - Maxparam procedure. }

MODULE Psub;
PROCEDURE Maxparam(VAR a:INTEGER; VAR b:INTEGER);

{ Two integer parameters are received by near reference. }
{ Near reference is specified with the VAR keyword. }

    BEGIN
        if a > b THEN
            b := a
        ELSE
            a := b
        END;
    END.
```

In the example, the C program adopts the Pascal naming convention and calling convention. Both programs must agree on whether parameters are passed by reference or by value; the following keywords affect the conventions:

- The `_pascal` keyword directs C to call Maxparam with the FORTRAN/Pascal naming convention (as MAXPARAM); `_pascal` also directs C to call Maxparam with the FORTRAN/Pascal calling convention.
- Since the procedure Maxparam can alter the value of either parameter, both parameters must be passed by reference. In this case, near reference is used; this method is specified in C by the use of near pointers, and in Pascal with the VAR keyword.

Far reference could have been specified by using far pointers in C. To specify far reference in Pascal, use the VARS keyword instead of VAR.

Calling a Pascal or PL/M Function from C

The following example demonstrates a C main module calling Pascal function fact. This function returns the factorial of an integer value.

```
/* C source file - calls Pascal function.
 * Compile in medium or large model. */

int _pascal fact(int n);

/* PASCAL keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Integer parameter passed by value. */

main()
{
    int x = 3;
    int y = 4;
    printf("The factorial of x    is %4d", fact(x));
    printf("The factorial of y    is %4d", fact(y));
    printf("The factorial of x+y is %4d", fact(x + y));
}

{ Pascal source code - factorial function. }
```

```
MODULE Pfun;
FUNCTION Fact (n : INTEGER) : INTEGER;
```

```
{Integer parameters received by value, the Pascal default. }

BEGIN
  Fact := 1;
  WHILE n > 0 DO
    BEGIN
      Fact := Fact * n;
      n := n - 1;      {Parameter n modified.}
    END;
  END;
END.
```

In the example, the C program adopts the Pascal naming convention and calling convention. Both programs must agree on whether parameters are passed by reference or by value. The `_pascal` keyword directs C to call `fact` with the FORTRAN/Pascal naming convention (as `FACT`); `_pascal` also directs C to call `fact` with the FORTRAN/Pascal calling convention.

The Pascal function `fact` should receive a parameter by value. Otherwise, the Pascal function will corrupt the parameter's value in the calling module. Passing by value is the default method for both C and Pascal.

C Calls to Assembly Language

In CTOS Microsoft C you can write assembly-language programs either by using the in-line assembler or by creating a stand-alone module using the CTOS Microsoft Macro Assembler (MASM). If you use the in-line assembler, you do not need to take any special precautions other than those outlined in the section Using the In-Line Assembler. The following paragraphs explain the techniques for interfacing your assembly-language routines with your C program.

When deciding whether to use the in-line assembler or MASM, there are several considerations. Here is a list of advantages MASM provides over the in-line assembler:

- MASM supports declaration of data in MASM format; in-line assembly does not.
- MASM has a more powerful macro capability than in-line assembly.
- Modules written for MASM can be interfaced more easily with modules written in more than one Microsoft high-level language.

- MASM assembles large assembly-language programs more quickly than the in-line assembler.
- MASM supports assembly-language code written prior to the existence of the in-line assembler.
- MASM error messages and warnings are more complete than those of the in-line assembler.

The in-line assembler is far more efficient for some assembly-language programming tasks. Here are some of the benefits of the in-line assembler:

- You can do spot optimizations by including short sections of assembly-language code in your C programs with the in-line assembler.
- Code written in in-line assembler does not necessarily incur the overhead of a function call; code assembled using MASM always does.
- You can include in-line assembly code in your C source files; code written for MASM must be in a separate file.

Writing the Assembly-Language Procedure

You must write your assembly-language procedure so that it uses the same calling conventions and naming conventions as your C program. If you follow these conventions, you will be able to write recursive procedures (procedures that call themselves), and be able to use the CodeView debugger to locate errors.

Note: *This subsection describes the simplified segment directives provided with the CTOS Microsoft Macro Assembler, version 5.1.*

The standard assembly-language interface method consists of these steps:

1. Setting up the procedure
2. Entering the procedure
3. Allocating local data (optional)
4. Preserving register values
5. Accessing parameters
6. Returning a value (optional)
7. Exiting the procedure

The subsections that follow describe each of these steps in detail.

Setting Up the Procedure

The linker cannot combine the assembly-language procedure with the C program unless you define compatible segments and declare the procedure properly. Perform the following steps to set up the procedure:

1. Use the `.MODEL` directive at the beginning of the source file; this directive automatically causes the appropriate kind of returns to be generated (NEAR for small or compact models; FAR for medium, large, or huge models).
2. Use the simplified segment directives `.CODE` and `.DATA` to declare the code and data segments.
3. Use the `PUBLIC` directive to declare the procedure label public. This declaration makes the procedure visible to other modules. Also declare any data you want to make public as `PUBLIC`.
4. Use the `EXTRN` directive to declare any global data or procedures accessed by the routine as external. The safest way to use `EXTRN` is to place the directive outside any segment definition; however, place near data inside the data segment.
5. Observe the C naming convention; precede all procedure names and global data names with an underscore.

Entering the Procedure

When you enter the procedure, in most cases you will want to set up a stack frame. This allows you to access parameters passed on the stack and to allocate local data on the stack. You do not need to set up the stack frame if your procedure accepts no arguments and does not use the stack.

To set up the stack frame, issue the instructions:

```
push    bp
mov     bp, sp
```

This sequence establishes BP as the frame pointer. You cannot use SP for this purpose because it is not an index or base register. Also, the value of SP may change as more data are pushed onto the stack. However, the value of the base register BP remains constant for the life of the procedure unless your program changes it, so each parameter can be addressed as an offset from BP.

The instruction sequence above preserves the value of BP, since it will be needed in the calling procedure as soon as your assembly-language procedure returns. It then transfers the value in SP to BP to establish a stack frame on entry to the procedure.

Allocating Local Data

Your assembly-language procedure can use the same technique for allocating temporary storage for local data that is used by high-level languages. To set up local data space, decrease the contents of SP just after setting up the stack frame. (To ensure correct execution, always increase or decrease SP by an even number.) Decreasing SP reserves space on the stack for local data. You must restore the space at the end of the procedure as follows:

```
push    bp
mov     bp, sp
sub     sp, space
```

In the example, space is the total size in bytes of the local data you want to allocate. Local variables are then accessed as fixed negative displacements from BP.

In the following example, the entry sequence establishes a stack frame and allocates temporary local storage for two words (4 bytes) of data. Later in the example, the program accesses the local storage, initializing both to 0.

```
push    bp           ; Save old stack frame.
mov     bp,sp        ; Set up new stack frame.
sub     sp,4         ; Allocate 4 bytes of local storage.
.
.
.
mov     WORD PTR [bp-2],0
mov     WORD PTR [bp-4],0
```

Note that local variables are also called dynamic, stack, or automatic variables.

Preserving Register Values

A procedure called from C should preserve the values of SI, DI, SS, and DS (in addition to BP, which is already saved). You should push any register value that your procedure modifies onto the stack after setting up the stack frame and allocating local storage, but prior to entering the main body of the procedure. Registers that your procedure does not alter need not be preserved.

Note: *Routines that your assembly-language procedure calls must not alter the SI, DI, SS, DS, or BP registers. If they do, and you have not preserved the registers, they can corrupt the calling program's register variables, segment registers, and stack frame, causing program failure. If your procedure modifies the direction flag using the STD or CLD instructions, you must preserve the flags register.*

This example shows an entry sequence that sets up a stack frame, allocates 4 bytes of local data space on the stack, then preserves the SI, DI, and flags registers.

```
push    bp           ; Save caller's stack frame.
mov     bp, sp       ; Establish new stack frame.
sub     sp, 4        ; Allocate local data space.
push    si           ; Save SI and DI registers.
push    di
pushf                   ; Save the flags register.
.
```

You must exit the example procedure with the following code:

```
popf                   ; Restore the flags register.
pop     di             ; Restore the old value in the DI
register.
pop     si             ; Restore the old value in the SI
register.
mov     sp, bp         ; Restore the stack pointer.
pop     bp             ; Restore the frame pointer.
ret                    ; Return to the calling routine.
```

If you do not issue the instructions above in the order shown, you will place incorrect data in registers. Follow the rules below when restoring the calling program's registers, stack pointer, and frame pointer:

- Pop all registers that you preserve in the reverse order from which they were pushed onto the stack. So, in the example, SI and DI are pushed, and DI and SI are popped.
- Restore the stack pointer by transferring the value of BP into SP before restoring the value of the frame pointer.
- Always restore the frame pointer last.

Accessing Parameters

Once you have established the frame pointer, allocated local storage (if required), and pushed any registers that need to be preserved, you can write the main body of the procedure. Figure 14-3 shows how functions that observe the C calling convention use the stack frame.

The stack frame for the assembly-language procedure shown in Figure 14-3 is established by the following:

1. The calling program pushes each of the parameters onto the stack, after which SP points to the last parameter pushed.
2. The calling program issues a CALL instruction, which causes the return address (the place in the calling program to which control will ultimately return) to be placed on the stack. This address can be either two bytes long (for near calls) or four bytes long (for far calls). SP now points to this address.
3. The first instruction of the called procedure saves the old value of BP, with the instruction `push bp`. SP now points to the saved copy of BP.
4. BP is used to hold the current value of SP, with the instruction `mov bp,sp`. BP therefore now points to the old value of BP (saved on the stack).
5. While BP remains constant throughout the procedure, SP is often decreased to provide room on the stack for local data or saved registers.

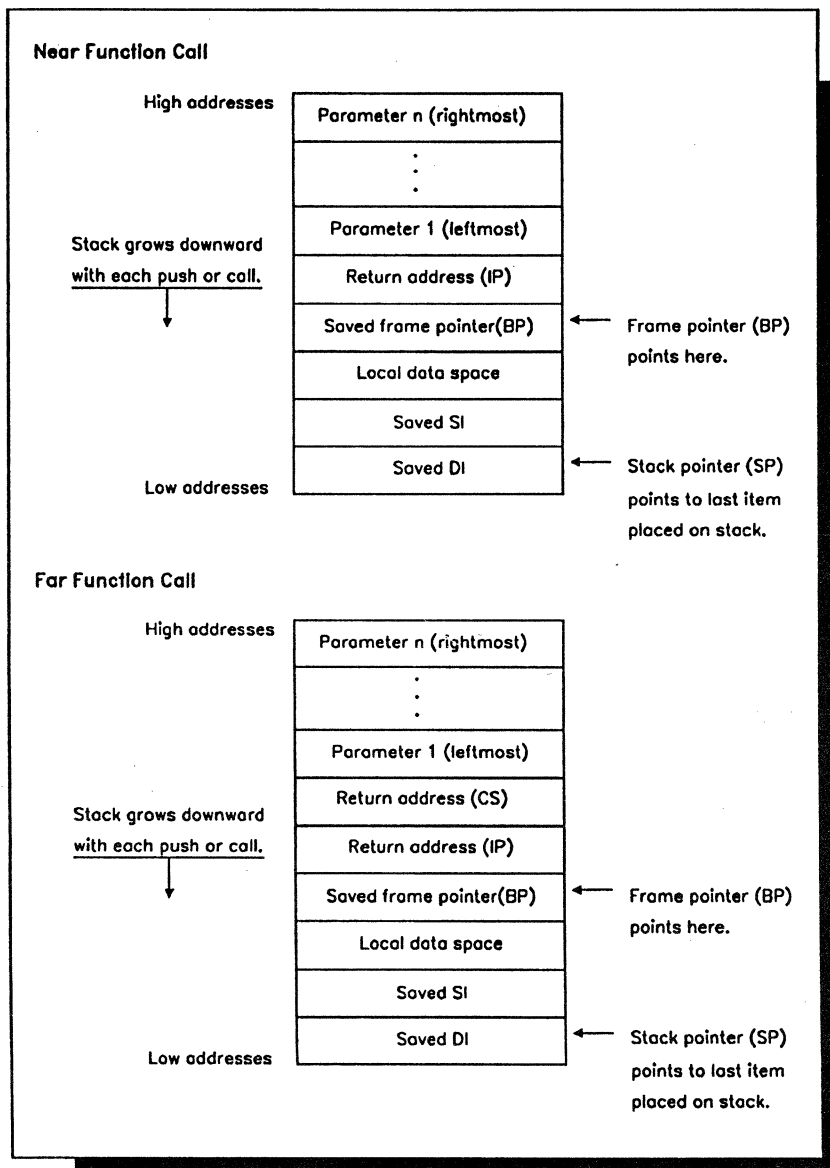


Figure 14-3. C Stack Frame

In general, the displacement (from BP) for a parameter *x* is equal to the size of return address plus 2 plus the total size of parameters between *x* and BP.

To calculate the size of parameters between *x* and BP, you must start with the rightmost parameter because C pushes parameters from right to left. For example, consider a FAR procedure that has one argument of type `int` (two bytes). The displacement of the parameter is:

$$\begin{aligned}\text{Argument's displacement} &= \text{size of far return address} + 2 \\ &= 4 + 2 \\ &= 6\end{aligned}$$

Thus, you can load the argument into BP with the following instruction:

```
mov     bx, [bp+6]
```

Once you determine the displacement of each parameter, you can use EQU directives or structures to refer to the parameter with a single identifier name in your assembly source code. For example, you can use a more readable name to reference the parameter at BP+6 if you put the following statement at the beginning of the assembly source file:

```
Arg1    EQU    [bp+6]
```

You can then refer to the first parameter in your source as `Arg1` in any instruction. Use of this feature is optional.

For far (segmented) addresses, C pushes the segment address before pushing the offset address. When pushing arguments larger than two bytes, high-order words are always pushed before low-order words, and parameters larger than two bytes are stored on the stack in most-significant, least-significant order.

This standard for pushing segment addresses before pushing offset addresses facilitates the use of the assembly-language instructions LDS (load data segment) and LES (load extra segment).

Returning a Value

Your assembly-language procedure can return a value to a C calling program. All return values of four bytes or less are passed in registers. Far pointers to return values larger than four bytes are returned in the DX and AX registers. The DX register contains the segment address; the AX register contains the offset relative to the segment contained in DX.

Table 14-4 shows the register conventions for returning simple data types to a C program.

Table 14-4. Register Conventions for Simple Return Values

Data Type	Registers
char	AL
int, short, _near *	AX
long, _far *	High-order portion (or segment address) in DX; low-order portion (or offset address) in AX

To return a structure from a procedure that uses the C calling convention, you must copy the structure to a global variable, then return a pointer to that variable in the AX register (DX:AX, if you compiled in compact, large, or huge model).

Procedures that use the FORTRAN/Pascal calling convention return structures similarly, with the following exceptions:

- The calling program allocates space for the return value on the stack.
- The calling program passes a pointer to the location where the return value is to be placed in a hidden parameter.
- Instead of copying your structure into a global data item, you copy it into the location pointed to by the hidden parameter.
- You must still return the pointer to that location in the AX register (or DX:AX for far data models).

Procedures that use the C calling convention and return type float or type double must always copy their return values into the global variable `_fac`. To return floating-point values from procedures declared with the FORTRAN/Pascal calling convention, you must return the result on the stack, just as you would a structure.

To return a value of type long double, you must place the value on the NDP(80x87) stack using the FLD instruction. The C run-time math routines guarantee that the only value on the NDP stack is a return value; your routines must observe the same rule.

Exiting the Procedure

Before you exit your assembly-language procedure, you must perform several steps to restore the calling program's environment. Some of these steps are dependent on actions you took in allocating space for local variables and preserving registers.

You must follow these steps (if appropriate to your procedure) in the order shown:

- If you saved any of the registers SS, DS, SI, or DI, they must be popped off the stack in the reverse order from which they were saved. If you pop these registers in any other order, your program will behave incorrectly.
- If you allocated local data space at the beginning of the procedure, you must restore SP with the instruction `mov sp, bp`.
- Restore BP with the instruction `pop bp`. This step is always necessary.
- Return to the calling program by issuing the `ret` instruction.

The following example shows the simplest possible entry and exit sequence. In the entry sequence, no registers are saved and no local data space is allocated.


```
push    bp
mov     bp, sp      ; Set up the new stack frame.
.
.
.
pop     bp          ; Restore the caller's stack frame.
ret
```

The following example shows an entry and exit sequence for a procedure that saves SI and DI and allocates local data space on the stack:

```
push    bp
mov     bp, sp      ; Establish local stack frame.
sub     sp, 4        ; Allocate space for local data.
push    si          ; Preserve the SI and DI registers.
push    di
.
.
.
pop     di          ; Pop saved registers.
pop     si
mov     sp, bp      ; Free local data space.
pop     bp          ; Restore old stack frame.
ret
```

Handling Data in Mixed-Language Programming

Text that follows covers these topics:

- Default Naming and Calling Conventions
- Numeric Data Representation
- Strings
- Array Declaration and Indexing
- Structures, Records, and User-Defined Types
- External Data
- Pointers and Address Variables
- Common Blocks
- Using a Variable Number of Parameters

Default Naming and Calling Conventions

Each language has its own default naming and calling conventions, as listed in Table 14-5.

Table 14-5. Default Naming and Calling Conventions

Language	Calling Convention	Naming Convention	Parameter Passing
C	C	Case sensitive	Value (scalar variables), reference (arrays and pointers)
FORTRAN	FORTRAN/Pascal	Case insensitive	Reference
Pascal	FORTRAN/Pascal	Case insensitive	Value

Pascal Conventions

You can modify the conventions observed by Pascal routines that interface with C functions by using the VAR, CONST, ADR, VARS, CONSTS, and ADRS keywords.

Numeric Data Representation

Table 14-6 shows how to declare numeric variables of similar type in different languages.

Table 14-6. Equivalent Numeric Data Types

C	FORTRAN	Pascal
short	INTEGER*2	INTEGER2
int	---	INTEGER (default)
unsigned short ¹	---	WORD
unsigned	---	---
long	INTEGER*4	INTEGER4
---	INTEGER (default)	---
unsigned long ¹	---	---
float	REAL*4	REAL4
---	REAL	REAL (default)
---	---	---
double	REAL*8	REAL8
---	DOUBLE PRECISION	---
long double	REAL*16	REAL16
unsigned char	CHARACTER*1 ²	CHAR

¹ Types unsigned short and unsigned long are not supported by FORTRAN. Type unsigned long is not supported by Pascal. A signed integral type can be substituted, but the maximum range will be less.

² The FORTRAN type CHARACTER*1 is not the same as LOGICAL.

The FORTRAN types COMPLEX*8 and COMPLEX*16 are not implemented in C but can be represented with structures.

The FORTRAN types LOGICAL*2 and LOGICAL*4 are not implemented in C. LOGICAL*2 is stored as a one-byte Boolean indicator followed by an unused byte; LOGICAL*4 is stored as a one-byte Boolean indicator followed by three unused bytes.

Strings

Each language implements strings differently. The following text describes the ways that strings are implemented in these Microsoft languages:

- C
- FORTRAN
- Pascal

C String Format

C stores strings as arrays of bytes and uses a null character ('\0') as an end-of-string delimiter. For example, consider the following string:

```
char c_string[ ] = "C text string";
```

This string is represented in memory as shown in Figure 14-4

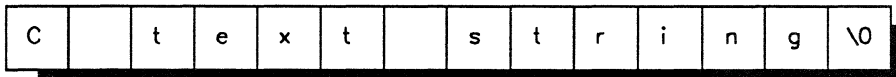


Figure 14-4. C String Format

Because `c_string` is an array like any other, C passes it by reference in function calls.

FORTRAN String Format

FORTRAN stores strings as a series of bytes at a fixed location in memory. There is no delimiter at the end of the string. Consider the string declared as follows:

```
STR = 'FORTRAN STRING'
```

The string is stored in memory as shown in Figure 14-5.

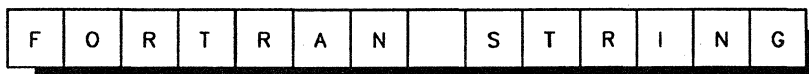


Figure 14-5. FORTRAN String Format

FORTRAN passes strings by reference, as it does all other data.

Note: *You cannot use FORTRAN's variable length strings in mixed-language programming because the temporary variable used to communicate string length is not accessible to other languages.*

To pass a C string to FORTRAN (or Pascal), pass the variable by reference as you normally would. In your FORTRAN or Pascal routine, you must specify the length of the string; strings that are passed as arguments from one language to another must be of fixed length.

Pascal String Format

Pascal represents strings as fixed-length arrays of CHAR or as strings with a length byte followed by the string data.

To pass a fixed-length string to a C function, use the concatenation operator (*) to append a null character. Then pass the string to the C function by reference (by declaring the string as CONST, CONSTS, VAR, or VARS). For example:

```
PROGRAM PasStr(input, output);
type
  stype15 = string(15); { fixed-length }
var
  str : stype15;

PROCEDURE PasStrToC(VAR s1 : stype15) [C]; EXTERN;
BEGIN
  str := 'Pass this to C' * chr(0);
  PasStrToC(str);
END.
```

A more flexible way to pass Pascal strings to C functions is to declare them as type ADRMEM or ADSMEM, then pass the address of the string. For example:

```
PROCEDURE PasStrToC(sladr : ADRMEM) [C]; EXTERN
```

Then you can call the C function with this code:

```
PasStrToC(ADR str);
```

Using this method, you can pass strings of different lengths to C functions.

Note: *The Pascal type LSTRING is not compatible with C; you can pass a string declared as LSTRING by first assigning it to another variable of type STRING, then passing that variable.*

Whenever you pass a variable of type STRING or type LSTRING by value, Pascal pushes the whole string onto the stack and passes the length of the string as another parameter. C cannot access strings passed in this manner.

Passing a string from a C function to a Pascal function or procedure is identical to passing a string from a C function to a FORTRAN routine. The only provision you must make is to specify the length of the string to your Pascal function.

Array Declaration and Indexing

Each language varies in the way that arrays are declared and indexed. Array indexing is a source-level consideration and involves no transformation of data. There are two differences in the way elements are indexed by each language:

- The value of the lower array bound is different among languages.
By default, FORTRAN indexes the first element of an array as 1. C indexes it as 0. Pascal lets you begin indexing at any integer value.
- Some languages vary subscripts in row-major order; others vary subscripts in column-major order.

This issue only affects arrays with more than one dimension. With row-major order (used by C and Pascal), the rightmost dimension changes first. With column-major order (used by FORTRAN), the leftmost dimension changes first. Thus, in C, the first four elements of an array declared as `X[3][3]` are:

`X[0][0]` `X[0][1]` `X[0][2]` `X[1][0]`

In FORTRAN, the four elements are:

`X(1,1)` `X(2,1)` `X(3,1)` `X(1,2)`

The C and FORTRAN arrays shown above illustrate the difference between row-major and column-major order as well as the difference in the assumed lower bound between C and FORTRAN. Table 14-7 shows equivalences for array declarations in each language. In this table, *r* is the number of elements of the row dimension (which changes most slowly), and *c* is the number of elements of the column dimension (which changes most quickly).

Table 14-7. Equivalent Array Declarations

Language	Array Declaration	Notes
C	<code>type x[r][c]</code>	When passed by reference
	<code>struct { type x[r][c]; } x</code>	When passed by value
FORTRAN	<code>type x(c, r)</code>	With default lower bounds of 1
Pascal	<code>x: ARRAY [a..a+r-1, b..b+c-1] OF type</code>	

The order of indexing extends to any number of dimensions you declare. For example, the C declaration:

```
int arr1[2][10][15][20];
```

is equivalent to the FORTRAN declaration:

```
INTEGER*2 ARR1(20, 15, 10, 2)
```

The constants used in a C array declaration represent dimensions, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as `int arr[5][5]` is `arr[4][4]`, not `arr[5][5]`.

Structures, Records, and User-Defined Types

The C struct type, the FORTRAN record (defined with the `STRUCTURE` keyword), and the Pascal record type are equivalent. Therefore, these data types can be passed between C, FORTRAN, and Pascal.

These types can be affected by the storage method. By default, C, FORTRAN, and Pascal use word alignment for types shorter than one word (type char and unsigned char). This storage method specifies that occasional bytes can be inserted as padding so that word and double-word objects start on an even boundary. (In addition, all nested structures and records start on a word boundary.)

If you are passing a structure or record across a mixed-language interface, your calling routine and called routine must agree on the storage method and parameter-passing convention. Otherwise, data will not be interpreted correctly.

Because Pascal, FORTRAN, and C use the same storage method for structures and records, you can interchange data between routines without taking any special precautions unless you modify the storage method. Make sure the storage methods agree before interchanging data between C, FORTRAN, and Pascal.

You can pass structures as parameters by value or by reference. Both the calling program and the called program must agree on the parameter-passing convention. For more information about the language you are using, refer to the heading Parameter-Passing Requirement.

External Data

External data refers to data that is both static and public; that is, the data is stored in a set place in memory as opposed to being allocated on the stack, and the data is visible to other modules.

External data can be defined in C, Pascal, and assembly language. Note that a data definition is distinct from an external declaration. A data definition causes a compiler to create a data object; an external declaration informs a compiler that the object is to be found in another module. FORTRAN can only define external data in COMMON blocks.

There are three requirements for programs that share external data between languages:

- One of the modules must define the data.

You can define a static data object in a C module by defining a data object outside all functions. (If you use the static keyword in C, however, the data object will not be made public.)

- The other modules that will access the data must declare the data as external.

In C, you can declare data as external by using an extern declaration, similar to the extern declaration for functions. In FORTRAN and Pascal, you can declare data as external by adding the EXTERN attribute to the data declaration.

- Resolve naming-convention differences.

In C, you can adopt the FORTRAN/Pascal naming convention by applying `_fortran`, `_plm`, or `_pascal` to the data declaration.

Pointers and Address Variables

Rather than passing data directly, you may want to pass the address of a piece of data. Passing the address amounts to passing the data by reference. In some cases, there is no other way to pass a data item as a parameter.

C programs always pass array variables by address. All other types are passed by value unless you use the address-of (&) operator to obtain the address.

The Pascal ADR and ADS types are equivalent to near and far pointers, respectively, in C. You can pass ADR and ADS variables as ADRMEM or ADSMEM. FORTRAN does not have formal address types, but does provide ways for storing and passing addresses.

Common Blocks

You can pass individual members of a FORTRAN or BASIC common block in an argument list, just as you can any data item. However, you can also give a different language module access to the entire common block at once.

C modules can reference the items of a common block by first declaring a structure with fields that correspond to the common-block variables. Having defined a structure with the appropriate fields, the C module must then connect with the common block itself. The next two subsections present methods for gaining access to common blocks.

Passing the Address of a Common Block

To pass the address of a common block, simply pass the address of the first variable in the block. (In other words, pass the first variable by reference.) The receiving C module should expect to receive a structure by reference.

In the following example, the C function `initcb` receives the address of the variable `N`, which it considers to be a pointer to a structure with three fields:

```
C      FORTRAN SOURCE CODE
C
      COMMON /CBLOCK/N, X, Y
      INTEGER*2 N
      REAL*8 X, Y
      .
      .
      .
      CALL INITCB(N)

/* C source code */

/* Explicitly set structure packing to word-alignment */
#pragma pack(2);
```

```
struct block_type
{
    int n;
    double x;
    double y;
};

initcb(struct block_type * block_hed)
{
    block_hed-n = 1;
    block_hed-x = 10.0;
    block_hed-y = 20.0;
}
```

Accessing Common Blocks Directly

You can access FORTRAN common blocks directly by defining a structure with the appropriate fields and then using the methods described under the heading External Data. Here is an example of accessing common blocks directly:

```
struct block_type
{
    int n;
    double x;
    double y;
};

extern struct block_type fortran cblock;
```

Using a Varying Number of Parameters

Some C functions (for example printf) accept a variable number of parameters. You cannot call such a function from another language.

**This Page is Blank
Do Not Print**

Section 15

Writing Portable Programs

Because C compilers exist for a variety of computers, some C applications can be ported to other systems. However, some aspects of language behavior depend on how specific computers operate, and on a specific C-compiler implementation. Thus, when designing portable programs, it is important to examine programming assumptions, especially those made regarding hardware and compiler dependencies.

This section describes programming assumptions to consider when writing portable programs, and covers the following topics:

- Hardware Assumptions
- Compiler Assumptions
- Portability of Data Files
- Portability Concerns Specific to CTOS Microsoft C
- CTOS Microsoft C Byte Ordering

The American National Standards Institute Standard for the C Language (the ANSI Standard) details every instance where language behavior is defined by the implementation. Appendix C summarizes implementation-defined behavior for CTOS Microsoft C.

Hardware Assumptions

Hardware assumptions fall into these categories:

- Size of Basic Data Types
- Storage Order and Alignment
- Byte Order in a Word
- Reading and Writing Structures
- Bit Fields in Structures
- Processor Arithmetic Mode
- Pointers
- Address Space
- Character Set

Size of Basic Data Types

Sizing issues affect these basic data types:

- `char`
- `int` and `short int`
- `float`, `double`, and `long double`

In C, the size of basic types is implementation-defined. Thus, because each implementation assumes specific sizes, you can not make assumptions about the size or alignment of data types within aggregate types. To write portable programs, use the `sizeof` operator to determine the size or amount of storage required for a variable or a type.

Table 15-1 summarizes the size of the basic types in CTOS Microsoft C.

Table 15-1. Size of Basic Types in CTOS Microsoft C

Type	Number of Bytes
char, unsigned char	1
int, short, unsigned int, unsigned short	2
near pointer	2
long, unsigned long	4
far pointer	4
float	4
double	8
long double	10

Type char

Type char is the smallest of the basic types, but it must be large enough to hold any of the characters in the implementation's basic character set. Normally, variables of type char are one byte.

Type int and Type short int

Type int and type short int often correspond to the register size of the target machine. Both int and short are greater than or equal to the size of type char but less than or equal to the size of type long.

If you assume that type `int` is a certain size, your code may not be portable for the following reasons:

- An `int` can be defined as either a 16-bit (two-byte) or a 32-bit quantity.
- An `int` is not always large enough to hold array indexes. For large arrays, you must use unsigned `int`; for extremely large arrays, use `long`. To be certain your code is portable, define your array indexes as `type_size_t`.

The file `LIMITS.H` contains manifest constants, listed below, for the maximum and minimum values of each basic integral type.

Constant	Value
<code>CHAR_BIT</code>	Number of bits in a variable of type <code>char</code>
<code>CHAR_MIN</code>	Minimum value a variable of type <code>char</code> can hold
<code>CHAR_MAX</code>	Maximum value a variable of type <code>char</code> can hold
<code>SCHAR_MIN</code>	Minimum value a variable of type signed <code>char</code> can hold
<code>SCHAR_MAX</code>	Maximum value a variable of type signed <code>char</code> can hold
<code>UCHAR_MAX</code>	Maximum value a variable of type unsigned <code>char</code> can hold
<code>SHRT_MIN</code>	Minimum value a variable of type <code>short</code> can hold
<code>SHRT_MAX</code>	Maximum value a variable of type <code>short</code> can hold
<code>USHRT_MAX</code>	Maximum value a variable of type unsigned <code>short</code> can hold

Constant	Value
INT_MIN	Minimum value a variable of type int can hold
INT_MAX	Maximum value a variable of type int can hold
UINT_MAX	Maximum value a variable of type unsigned int can hold
LONG_MIN	Minimum value a variable of type long can hold
LONG_MAX	Maximum value a variable of type long can hold
ULONG_MAX	Maximum value a variable of type unsigned long can hold

Type float, Type double, and Type long double

Type float is the smallest of the basic floating-point types. Type double is usually larger than type float, and type long double is usually the largest of the floating-point types. You can make only the following portability assumptions about floating-point types:

- Any value that can be represented as type float can be represented as type double (type float is a subset of type double).
- Any value that can be represented as type double can be represented as type long double (type double is a subset of type long double).

The file `FLOAT.H` contains manifest constants, listed below, for the maximum and minimum values of each basic floating-point type.

Constant	Value
DBL_DIG	Number of decimal digits of precision a variable of type double can hold
DBL_MAX	Maximum value a variable of type double can hold
DBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type double can hold
DBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type double can hold
DBL_MIN	Minimum positive value a variable of type double can hold
DBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type double can hold
DBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type double can hold
FLT_DIG	Number of decimal digits of precision a variable of type float can hold
FLT_MAX	Maximum value a variable of type float can hold
FLT_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type float can hold
FLT_MAX_EXP	Maximum value (base 2) the exponent of a variable of type float can hold
FLT_MIN	Minimum positive value a variable of type float can hold
FLT_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type float can hold

Constant	Value
FLT_MIN_EXP	Minimum value (base 2) the exponent of a variable of type float can hold
LDBL_DIG	Number of decimal digits of precision a variable of type long double can hold
LDBL_MAX	Maximum value a variable of type long double can hold
LDBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type long double can hold
LDBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type long double can hold
LDBL_MIN	Minimum positive value a variable of type long double can hold
LDBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type long double can hold
LDBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type long double can hold

Storage Order and Alignment

The C language does not define any specific layout for the storage of data items relative to one another. The layout for storage of structure elements, or unions within a structure or union, is defined by the implementation.

Thus, order and alignment issues affect these data types:

- Structures
- Unions

Some processors require that data longer than one byte be word-aligned (aligned to an even-byte address). Other processors, such as the 80x86 family, do not have such a restriction.

Structure Order and Alignment

The following example illustrates how alignment can affect your program. In the example, a structure is cast to type long because the programmer knew the order in which a particular implementation stored data.

```
/* Nonportable code */
struct time
{
    char hour;          /* 0 < hour < 24 -- fits in a char */
    char minute;        /* 0 < minute < 60 -- fits in a char */
    char second;        /* 0 < second < 60 -- fits in a char */
};

.
.
.
struct time now, alarm time;

.
.
.
if ((long)now >= (long)alarm time)
{
    /* sound an alarm */
}
```

The preceding code makes these nonportable assumptions:

- The data for hour will be stored in a higher order position than minute or second. Because C does not guarantee storage order or alignment of structures or unions, the code may not be portable to other machines.
- Three variables of type char will be shorter than or the same length as a variable of type long. Thus, the code is not portable according to the rules governing the size of basic types.

If either of these assumptions proves false, the comparison (if statement) is invalid.

To make the program in the preceding example portable, you can break the comparison between the two long integers into a component-by-component comparison. This technique is illustrated in the following example:

```
/* Portable code */
struct time
{
    char hour; /* 0 < hour < 24 -- fits in a char */
    char minute; /* 0 < minute < 60 -- fits in a char */
    char second; /* 0 < second < 60 -- fits in a char */
};

.
.
.
struct time now, alarm_time;
.
.
.
if (time_cmp(now, alarm_time) >= 0)
{
    /* sound an alarm */
}
.
.
.
int time_cmp(struct time t1, struct time t2)
{
    if(t1.hour != t2.hour)
        return(t2.hour - t1.hour);
    if(t1.minute != t2.minute)
        return(t2.minute - t1.minute);
    return(t2.second - t1.second);
}
```

Union Order and Alignment

Programmers use unions most often for two purposes: to store data types not known until run time, and to access the same data in different ways.

Unions falling into the second category are usually not portable. For example, the union below is not portable:

```
union tag_u
{
    char bytes_in_long[4];
    long a_long;
};
```

The intent of the union above is to access the individual bytes of a variable of type long. However, the union may not work as intended when ported to other computers because:

- It relies on a constant size for type long.
- It may assume byte ordering within a variable of type long.

The first problem can be addressed by coding the union as follows:

```
union tag_u
{
    char bytes_in_long[sizeof(long) / sizeof(char)];
    long a_long;
};
```

Note the use of the sizeof operator to determine the size of a data type.

Byte Order in a Word

The order of bytes within a word (int or short) or a double word (long) can vary among machines. Code that assumes an internal order is not portable, as shown by this example:

```
/* Nonportable structure to access an int in bytes. */
struct tag_int_bytes
{
    char lobyte;
    char hibyte;
};
```

A more portable way to access the individual bytes in a word is to define two macros that rely on the constant `CHAR_BIT`, defined in `LIMITS.H`:

```
#define LOBYTE(a) (char)((a) & 0xff)
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

The `LOBYTE` macro is still not completely portable. It assumes that a char is eight bits long, and it uses the constant `0xff` to mask the high-order eight bits. Because portable programs cannot rely on a given number of bits in a byte, consider the revision below:

```
#define LOBYTE(a) (char)((a) & ((unsigned)~ 0>>CHAR_BIT))
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

The new `LOBYTE` macro performs a bitwise complement on 0; that is, all zero bits are turned into ones. It then takes that unsigned quantity and shifts it right far enough to create a mask of the correct length for the implementation.

The following code assumes that the order of bytes in a word will be least-significant first:

```
int c;
.
.
.
fread(&c, sizeof(char), 1, fp);
```

The code attempts to read one byte as an int, without converting it from a char. However, the code will fail in any implementation where the low-order byte is not the first byte of an int. The following solution is more portable. In the example below, the data is read into an intermediate variable of type char before being assigned to the integer variable.

```
int c;
char ch;
.
.
.
fread(&ch, sizeof(char), 1, fp);
c = ch;
```


The following example shows how to use the C run-time function `fgetc` to return the value. The `fgetc` function returns type `char`, but the value is promoted to type `int` when it is assigned to a variable of type `int`.

```
int c;  
.  
.  
.  
c = fgetc(fp);
```

Storage Alignment

For improved performance, CTOS Microsoft C normally aligns data types longer than one byte to an even-byte address. For information about controlling structure packing, refer to the `/Zp` compiler option and the `pack pragma` in online Help.

Reading and Writing Structures

Many C programs read data from disk into structures and write data to disk from structures. The functions that perform disk I/O in C require you to specify the number of bytes to be transferred.

When performing disk input and output, structures may be different sizes depending on the selected structure-packing option. Thus, you should always use the `sizeof` operator to obtain the size of the data to be read or written because differing data type sizes or alignment schemes may alter the size of a given structure. For example:

```
fread(&my_struct, sizeof(my_struct), 1, fp);
```

Bit Fields in Structures

The CTOS Microsoft C compiler implements bit fields, while many C compilers do not. Bit fields allow you to access individual bits within a data item.

While the practice of accessing bits is inherently nonportable, you can improve your chances of porting a program that uses bit fields if you make no assumptions about these aspects of bit fields:

- Order of Assignment
- Size and Alignment of Bit Fields

Order of Assignment

Bit fields order of assignment in memory is left to each implementation. Thus, in a bit field structure, you cannot rely on particular entries being in a higher order position than other entries.

This problem is similar to the portability constraint imposed by alignment of basic data types in structures. The C language does not define any specific layout for the storage of data items relative to one another.

Size and Alignment of Bit Fields

CTOS Microsoft C supports bit fields up to the size of type long. Each individual member of the bit field structure can be up to the size of the declared type. Some compilers do not support bit field-structure elements that are longer than type int.

The example below defines a bit field, `short_bitfield`, that is shorter than type int:

```
struct short_bitfield
{
    unsigned usr_bkup : 1; /* 0 <= usr_bkup < 1 */
    unsigned usr_sec : 4; /* 9 <= usr_sec < 16 */
};
```

The following example defines a bit field, `long_bitfield`, that has elements longer than type int:

```
struct long_bitfield
{
    unsigned long disk_pos : 22; /* 0 <= disk_pos < 4,194,304 */
    unsigned long rec_no : 10; /* 0 <= rec_no < 1,024 */
};
```

The bit field `short_bitfield` is likely to be supported by more implementations than `long_bitfield`.

CTOS Microsoft C Specific

The example below introduces another portability issue: alignment of data defined in bit fields. CTOS Microsoft C does not allow an element in a structure to extend across two words. The first two elements, day and month, take up nine bits. The third, year, would extend across a word boundary, so it must begin on the next word boundary.

```
struct long_bitfield
{
    unsigned int day : 5;      /* 0 <= day < 32 */
    unsigned int month : 4;    /* 0 <= month < 16 */
    unsigned int year : 11;    /* 0 <= year < 2048 */
};
```

Figure 15-1 illustrates the example above.

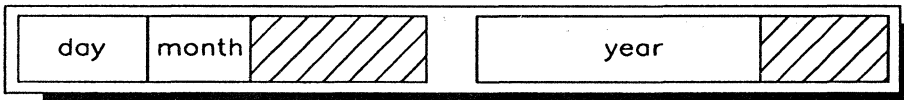


Figure 15-1. Data Alignment in Bit Fields

Other compilers may not use the same storage techniques.

Processor Arithmetic Mode

Two types of arithmetic are common on digital computers:

- One's Complement
- Two's Complement

CTOS Microsoft C produces code for the Intel 80x86 processors only, which all perform two's-complement arithmetic. However, some programs assume that all target computers perform two's-complement arithmetic.

If you take advantage of the fact that a given operation causes a particular bit pattern to be set on either a one's-complement or two's-complement computer, your program will not be portable. For example, two's-complement machines represent the eight-bit integer value -1 as a binary 11111111. A one's-complement machine represents the same decimal value (-1) as 11111110.

Some programmers assume that -1 will fill a byte or a word with ones, and use it to construct a mask template that they shift later. This will not work correctly on one's-complement machines, but the error will not surface until the least-significant bit is used.

In two's-complement arithmetic, there is only one value that represents zero. In one's-complement arithmetic, there is a value for zero and a value for negative zero. To handle this anomaly correctly, use the C relational operators. If you write code that deliberately circumvents the C relational operators, tests for zero or NULL may not operate correctly.

Pointers

One of the most powerful but potentially dangerous features of the C language is its use of indirect addressing through pointers. Bugs introduced by misusing pointers can be difficult to detect and isolate because the error often corrupts memory unpredictably.

The following issues affect portability of pointers:

- Casting
- Size
- Subtraction
- Null Values

Casting Pointers

When casting pointers to different data types, be sure not to make nonportable assumptions.

```
/* Nonportable coercion */
char c[4];
long *lp;

lp = (long *)c;
*lp = 0x12345678L;
```

This code is nonportable because using a cast to change an array of char to a pointer of type long assumes a particular byte-ordering scheme.

Pointer Size

A pointer can be assigned (or cast) to any integer type large enough to hold it, but the size of the integer type depends on the machine and the implementation. (In fact, it can even depend on the memory model.) Therefore, you cannot assume:

```
sizeof(char *) == sizeof(int)
```

To determine the size of any unmodified data pointer, use the size of a generic data pointer:

```
sizeof(void *)
```

Pointer Subtraction

Code that assumes pointer subtraction yields an int value is nonportable. Pointer subtraction yields a result of type `ptrdiff_t` (defined in `STDDEF.H`). Portable code must always use variables of type `ptrdiff_t` for storing the result of pointer subtraction.

Null Pointers

In most implementations, NULL is defined as 0. In CTOS Microsoft C, it is defined as `((void *)0)`. Because code pointers and data pointers are often different sizes, using 0 for the null pointer for both can lead to nonportability. The difference in size between code pointers and data pointers will cause problems for functions that expect pointer arguments longer than an int.

To avoid these problems, either use the null pointer, as defined in the include file `STDDEF.H`; use prototypes; or explicitly cast NULL to the correct data type. The following example shows a portable way to use the null pointer:

```
/* Portable use of the NULL pointer */
main()
{
    func

    ((char *)NULL);
    func2((void *(*()) )NULL);
}

void func1(char * c)
{
}

void func2(void *(* func)())
{
}
```

The invocations of `func1` and `func2` explicitly cast NULL to the correct size. In the case of `func1`, NULL is cast to type `char *`; in the case of `func2`, it is cast to a pointer to a function that returns type `void`.

CTOS Microsoft C Specific

Subtraction of pointers to huge arrays that have more than 32,767 elements may yield a long result. The `_huge` keyword is implementation-defined by CTOS Microsoft C and is not portable. Here is how to subtract pointers to huge arrays:

```
char _huge *a;
char _huge *b;
long d;
.
.
.
d = (long)(a - b);
```

This technique is also appropriate for subtraction of char pointers to character arrays which are larger than 32,767 bytes.

In CTOS Microsoft C, the memory model selected and the special keywords `_near`, `_far`, and `_huge` can change the size of a pointer. The Microsoft C memory models and extended keywords are nonportable, but you should be aware of their effects.

Sizes of generic pointers and default pointer sizes are shown in Tables 15-2 and 15-3, respectively.

Table 15-2. Size of Generic Pointers

Declaration	Name	Size
<code>void _near *</code>	Generic near pointer	16 bits
<code>void _far *</code>	Generic far pointer	32 bits
<code>void _huge *</code>	Generic huge pointer	32 bits

Table 15-3. Default Pointer Sizes

Memory Model	Code Pointer Size	Data Pointer Size
Small	16 bits	16 bits
Medium	32 bits	16 bits
Compact	16 bits	32 bits
Large	32 bits	32 bits
Huge	32 bits	32 bits

Address Space

The amount of available memory and the address space on systems vary, depending on many factors outside your control. However, a program designed with portability in mind should handle insufficient-memory situations.

To ensure that your program handles these situations, you should always check the error return from any of the dynamic memory allocation routines, such as `malloc`, `calloc`, `strdup`, and `realloc`. Insufficient memory situations occur not only because of a lack of installed memory but also because too many other applications are using memory.

Character Set

The C language does not define the character set used in an implementation. This means that any programs that assume the ASCII character set are nonportable.

These are the only restrictions on the character set:

- No character in the implementation's character set may be larger than the size of type `char`.
- Each character in the set must be represented as a positive value by type `char`, whether it is treated as signed or unsigned. So, in the case of the ASCII character set and an eight-bit `char`, the maximum value is 127 (128 is a negative number when stored in a `char` variable).

In addition to these restrictions, two issues affect character set manipulation:

- Character Classification
- Case Translation

Character Classification

The standard C run-time support contains a complete set of character-classification macros and functions. These functions are defined in the `CTYPE.H` file and are guaranteed to be portable:

<code>isalnum</code>	<code>isalpha</code>	<code>isctrl</code>	<code>isdigit</code>
<code>isgraph</code>	<code>islower</code>	<code>isprint</code>	<code>ispunct</code>
<code>isspace</code>	<code>isupper</code>	<code>isxdigit</code>	

The following code fragment is not portable to implementations that do not use the ASCII character set:

```
/* Nonportable */
if(c >= 'A' && c <= 'Z')
/* uppercase alphabetic */
Instead, consider using this:

/* Portable */
if(isalpha(c) && isupper(c))

    /* uppercase alphabetic */
```

The first example is nonportable because it assumes that uppercase A is represented by a smaller value than uppercase Z, and no lowercase characters fall between the values of A and Z. The second example is portable because it uses the character classification functions to perform the tests.

In a portable program, you should not perform any comparison on variables of type `char` except strict equality (`==`). You cannot assume the character set follows an increasing sequence--that may not be true on a different machine.

Case Translation

Translation of characters from upper- to lowercase or from lower- to uppercase is called case translation. The following example shows a coding technique for case translation not portable to implementations using a non-ASCII character set.

```
#define make_upper(c) ((c)&0xcf)
#define make_lower(c) ((c)|0x20)
```

This code takes advantage of the fact that you can map uppercase to lowercase simply by changing the state of bit 6. It is extremely efficient but nonportable. To write portable code, use the case-translation macros `toupper` and `tolower` (defined in `CTYPE.H`).

Assumptions about the Compiler

Different compilers translate C source code into object code in different ways. The ANSI standard for the C programming language defines the way that many of these translations must be done. Other translations are implementation-defined.

This subsection describes compiler assumptions that can make your programs nonportable. These assumptions fall into the following categories:

- Sign Extension
- Length and Case of Identifiers
- Register Variables
- Functions with a Variable Number of Arguments
- Evaluation Order
- Function and Macro Arguments with Side Effects
- Environment Differences

Sign Extension

Sign extension is the propagation of the sign bit to fill unoccupied space when promoting to a more-significant type, or when performing bitwise right-shift operations. Issues relating to sign extension fall into these categories:

- Promotion from Shorter Types
- Bitwise Right-Shift Operations

Promotion from Shorter Types

Integral promotions from shorter types occur when you make an assignment, perform arithmetic, perform a comparison, or perform an explicit cast.

The behavior of integral promotion is well defined, except for type `char`. The implementation defines whether type `char` is treated as signed or unsigned.

CTOS Microsoft C allows you to treat type `char` as signed or unsigned. By default, a `char` is considered signed, but if you change the default `char` type using the `/J` compiler option, you can treat it as unsigned.

The code fragment below is an example of promotion as a result of assignment:

```
char c1 = -3;
int i1;

i1 = c1;
```

In this example, the expected result of the assignment statement is that `i1` will be set to `-3`. If the implementation defines type `char` as unsigned, however, sign extension will not occur, and `i1` will be `253` (on a two's-complement machine).

Promotion can also occur as a result of a comparison of different types:

```
char c;

if(c == 0x80)
    .
    .
    .
```

This comparison will never evaluate as true on an implementation that sign-extends `char` types but treats hexadecimal constants as unsigned. Use a character constant of the form `'\x80'`, or explicitly cast the constant to type `char` to perform the comparison correctly.

The following comparison, which is an example of promotion as a result of a cast, is also nonportable:

```
char c;
unsigned int u;

if(u == (unsigned)c)
```

This code illustrates two problems:

- The char type may be treated as signed or unsigned, depending on the implementation.
- If the char type is treated as signed, it can be converted to unsigned in two different ways: the char value may first be sign-extended to int, then converted to unsigned; or the char may be converted to unsigned char, then sign-extended to int length.

It is always safe to compare a signed int with a char constant because C requires all character constants to be positive.

Variables of type char are promoted to type int when passed as arguments to a function. This will cause sign extension on some machines. Consider the following code:

```
char c = 128;
printf("%d\\ n", c);
```

Bitwise Right-Shift Operations

Positive or unsigned integral types (char, short, int, and long) yield positive or zero values after a right bitwise shift (>>) operation. For example:

Expression	Yields
(char)120 >> 4	7
(unsigned char)240 >> 8	0
(int)500 >> 8	1
(unsigned int)65535 >> 4	4,095

Negative-signed integral types yield implementation-defined values after a bitwise right-shift operation. This means that you must know whether you want to do a signed or unsigned shift, then code accordingly.

If you do not know how the implementation performs, you may get unexpected results. For example, `(signed char)0x80 >> 3` yields `0xf0` if the implementation performs sign extension on right bitwise shifts. If the implementation does not perform the sign extension, the result is `0x10`.

You can use right shifts to speed up division when the divisor can be represented by powers of 2 and the dividend is positive. To maintain portability, you should use the division operator.

To perform an unsigned shift, explicitly cast the data to an unsigned type. To perform a shift that extends the sign bit, use the division operator as follows: divide by 2^n , where n is the number of bits you want to shift.

Length and Case of Identifiers

All implementations do not support long identifiers: some allow 6 characters only; others allow up to 32. They may report each identifier that exceeds the maximum length or truncate identifiers to a given length. Truncation causes serious problems, especially if you have a number of similarly named variables within the scope of a block of code, such as the following:

```
double acct_receivable_30_days;  
double acct_receivable_60_days;  
double acct_receivable_90_days;  
double current_interest_rate;  
  
acct_receivable_days *= current_interest_rate;
```

If your target system retains only six significant characters, you will have to rename all your `acct_receivable` variables.

Case sensitivity also affects portability. C is usually a case-sensitive language. That is, `CalculateInterest` is not considered the same identifier as `calculateinterest`. Some systems are not case sensitive, however, so to write portable code, differentiate your identifiers by something other than case.

These problems with identifiers can occur in two locations: the compiler and the linker or loader. Even if the compiler can handle long and case-differentiated identifiers, if the linker or loader cannot, you can get duplicate definitions or other unexpected errors.

Register Variables

The number and type of register variables in a function depend on the implementation. You can declare more variables as register than the number of physical registers the implementation uses. In such a case, the compiler treats the excess register variables as automatic.

Since the types that qualify for register class differ among implementations, invalid register declarations are treated as automatic.

If you declare variables as register to optimize performance, declare them in decreasing order of importance to ensure that the compiler allocates a register to the most important variables.

CTOS Microsoft C Specific

The compiler ignores register declarations if you select the global register allocation optimization. You can select global register allocation as follows:

Environment	Selection
CL command line	Specify either the <code>/Oe</code> or <code>/Ox</code> option.
PWB	Select the Global Register Allocation option in the Debug Build Options or Release Build Options dialog boxes.
pragma	Use the optimize pragma with the <code>e</code> parameter.

The compiler suppresses the use of register variables if you use the `/r` option.

Functions with a Variable Number of Arguments

Functions that accept a variable number of arguments are not portable. Although the ANSI Standard specifies how to write these functions and how they behave, differences still exist among compiler implementors about how to use variable argument lists.

Many UNIX systems support a standard that differs from the ANSI Standard for variable arguments. Although this may change, it currently presents a portability concern.

CTOS Microsoft C run-time libraries and macros allow you to use whichever version of variable argument support you expect to be most portable for your application.

Evaluation Order

The C language does not guarantee the evaluation order of most expressions. Avoid writing constructs that depend on evaluation within an expression to proceed in a particular manner. For example:

```
i = 0;
func(i++, i++);
.
.
.
func(int b, int c)
{
```

A compiler could evaluate this code fragment and pass 0 as b and 1 as c. It could also pass 1 as b and 0 as c and conform equally with the standards.

The C language does guarantee that an expression will be completely evaluated at any given sequence point. A sequence point is a point in the syntax of the language at which all side effects of an expression or series of expressions have been completed.

These are the sequence points in the C language:

- The semicolon (;) statement separator
- The call to a function after the arguments have been evaluated
- The end of the first operand of one of the following:
 - Logical AND (&&)
 - Logical OR (||)
 - Conditional (?)
 - Comma separator (,) when used to separate statements or in expressions; the comma separator is not a sequence point when it is used between variables in declaration statements or between parameters in a function invocation
- The end of a full expression, such as:
 - An initializer
 - The expression in an expression statement (for example, any expression inside parentheses)
 - The controlling expression of a while or do statement
 - Any of the three expressions of a for statement
 - The expression in a return statement

Function and Macro Arguments with Side Effects

Run-time support functions can be implemented either as functions or as macros. Avoid including expressions with side effects inside function invocations unless you are sure the function will not be implemented as a macro. Here is an illustration of how an argument with side effects can cause problems:

```
#define limit_number(a) ((a>1000)?1000:(a))  
a = limit_number(a++);
```

If $a \leq 1000$, it is incremented once. If $a > 1000$, it is incremented twice, which is probably not the intended behavior.

A macro can be used safely with an argument that has side effects if it evaluates its parameter only once. You can determine whether a macro is safe only by inspecting the code.

A common example of a run-time support function that is often implemented as a macro is `toupper`. You will find your program's behavior confusing if you use the following code:

```
char c;  
c = toupper(getc());
```

If `toupper` is implemented as a function, `getc` is called only once, and its return value is translated to uppercase. However, if `toupper` is implemented as a macro, `getc` is called once or twice, depending on whether `c` is upper- or lowercase. Consider the following macro example:

```
#define toupper(c) ((islower(c)) ? _toupper(c) : (c))
```

If you include the `toupper` macro in your code, the preprocessor expands it as follows:

```
/* What you wrote */  
c = toupper(getc());  
  
/* Macro expansion */  
ch = (islower((getc())) ? _toupper(getc()) : (getc()));
```

The expansion of the macro shows that the argument to `toupper` will always be called twice: once to determine if the character is lowercase and the next time to perform case translation (if necessary). In the example, this double evaluation calls the `getc` function twice. Because `getc` is a function whose side effect is to read a character from the standard input device, the example requests two characters from standard input.

Environment Differences

Many programs perform some file I/O. When writing these programs for portability, consider the following:

- Do not hard-code file or path names. Use constants you define either in a header file or at the beginning of the program.
- Do not assume the use of any particular file system. For example, the UNIX-model, hierarchical file system is prevalent on small computers. On larger systems, the file system often follows a different model.
- Do not assume a particular display size (number of rows and columns).
- Do not assume that display attributes exist. Some environments do not support such attributes as color, underlined text, blinking text, highlighted text, reverse text, protected text, or dim text.

Portability of Data Files

Data files are rarely portable across different CPUs. Structures, unions, and arrays have varying internal layout and alignment requirements on different machines. In addition, byte ordering within words and actual word length may vary.

The best way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data is written and read as characters. The only portability problem you are likely to encounter if you follow this course is a conflict in character sets; many computers have character-set conversion utilities.

Portability Concerns Specific to CTOS Microsoft C

CTOS Microsoft C offers extensions that let you take advantage of the full capabilities of Intel 80x86 processors. These extensions are not portable to other compilers or environments. The following list shows keywords specific to CTOS Microsoft C:

<code>_asm</code>	<code>_far</code>	<code>_interrupt</code>	<code>_plm</code>
<code>_based</code>	<code>_fastcall</code>	<code>near</code>	<code>_saveregs</code>
<code>cdecl</code>	<code>fortran</code>	<code>_near</code>	<code>_segment</code>
<code>_cdecl</code>	<code>_fortran</code>	<code>_loadds</code>	<code>_segname</code>
<code>_export</code>	<code>huge</code>	<code>pascal</code>	
<code>far</code>	<code>_huge</code>	<code>_pascal</code>	

CTOS Microsoft C Byte Ordering

Tables 15-4 and 15-5 summarize CTOS Microsoft C byte ordering for short and long types, respectively. In these tables, the least-significant byte of the data item is b0; the next byte is denoted by b1, and so on.

Since byte ordering is machine specific, any program that uses this byte ordering is not portable.

Table 15-4. Byte Ordering for Short Types

CPU	Byte Order
8086	b0 b1
80286	b0 b1
PDP-11	b0 b1
VAX-11	b0 b1
M68000	b1 b0
Z8000	b1 b0

Table 15-5. Byte Ordering for Long Types

CPU	Byte Order
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
PDP-11	b2 b3 b0 b1
VAX-11	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000	b3 b2 b1 b0

Section 16

Building CTOS III Applications

Using CTOS Microsoft C, you can create applications for CTOS III. This section tells you how to create:

- multithread applications
- dynamic-link libraries that multiple applications can use

Creating Multithread CTOS III Applications

CTOS Microsoft C provides support for creating multithread applications under CTOS III. You should consider using more than one thread if your application needs to manage multiple activities. This subsection explains the features in CTOS Microsoft C 6.1 that support the creation of multithread programs.

Multithread Programs

A thread is basically a path of execution through a program. It is also the smallest unit of execution that CTOS III schedules. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all of the task's resources.

A thread is not the same as a traditional CTOS process. A thread is a special class of CTOS III process characterized as follows:

- The C source code for the thread must be compiled with the `/MT` compiler option.
- `LLIBCMT.LIB`, the multithreaded version of the C Library, must be used when linking a runfile containing threads. `LLIBCMT.LIB` must be used with CTOS III's imports library, `SystemImp.Lib`, which establishes a multithreaded runfile as a client of CTOS III's `system.dll`.
- A thread's executions must be initiated by calling the `_beginthread` function.

A task consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority may have to wait while higher priority threads complete their tasks.

All threads in a task operate independently of one another. Unless you take special steps to make them visible to each other, each thread executes while completely unaware of the existence of other threads in a process. Threads sharing common resources, however, must coordinate their work by using flags, semaphores or some other method of interprocess communication. Refer to *Writing a Multithread Program* for more information about synchronizing threads.

Library Support

If one thread is suspended while executing a function, one of the program's other threads might start executing. If the second thread calls the same function, data might be corrupted. To avoid this, access to static data used by the function must be restricted to one thread at a time. This process of restricting access to certain data is called **serialization**.

You do not need to serialize access to stack-based (automatic) variables because each thread has a different stack. Therefore, a function that uses only automatic (stack) variables is re-entrant. The standard C run-time libraries have a limited number of re-entrant functions (listed in Appendix F). A multithread program needing to use C run-time library functions that are not re-entrant should be built with the multithread library `LLIBCMT.LIB`.

The Multithread C Library `LLIBCMT.LIB`

`LLIBCMT.LIB` is a re-entrant large-model library for creating multithread programs.

All calls to library functions must use the large-model calling interface (far code pointers, far calls, and far data pointers). When your application calls functions in this library, the following are true:

- All library calls must be far calls.
- All library calls must use the C calling convention; programs compiled using the `/Gr` (fastcall calling convention) or `/Gc` (Pascal calling convention) options must use the standard include files for the run-time library functions they call.
- All data and code pointers must be far pointers.
- Variables passed to library functions must either be passed by value or cast to a far address.
- Your main function must be declared far if you are compiling with the small or compact memory models.

You do not need to explicitly declare far pointers if you are using the compact, large, or huge memory models, since these models use far pointers as default. For the large and huge memory models, the function calls are also far by default.

A small-model program calling a library function such as `isupper`, for example, must use declarations like the following:

```
int _far _cdecl isupper(int _c)
```

LLIBCMT.LIB cannot be used with the C Library Extension library, **LIBCEXT.LIB**, but must be linked with CTOS III's **SystemImp.Lib**, the import library for **system.dll**.

Programs built with **LLIBCMT.LIB** do not share C run-time library code or data with any dynamic-link libraries they call. Refer to **Dynamic Linking with CTOS III** for information on how to build DLLs and how to share code and data between tasks.

The Multithread Library Compile Option (/MT)

The **/MT** option for the **CL** command is the best way to build a multithread program with **LLIBCMT.LIB**. Using the **/MT** option automatically specifies the **/ALw /FPi /G2 /D MT** options.

The following table describes the effects of these options:

Table 16-1. Multithread Options

Switch	Effect
/ALw	Use the large memory model with separate stack segment; do not reload the DS register as part of the entry sequence for every function
/FPi	Generate in-line floating-point instructions and select the emulator math package
/G2	Use the 80286 processor instruction set
/D MT	Use the multithread version of the definitions in the include files

These options can be combined with other options to specify different memory models and different relationships between the data segment and the stack. You can override the /G2 and /FPi options by specifying the desired option after /MT on the command line.

The following example shows how to override the floating-point package option:

```
CL  
[Args] /MT /FPa /Lp PROG.C
```

Note: *You cannot replace the /MT option with /ALw /FPi /G2. You must use /MT to generate multithread programs.*

Include Files

The CTOS Microsoft C include files contain conditional sections for multithread applications using LLIBCMT.LIB. To compile your application with the appropriate definitions, you can:

- compile with the /MT option described in Library Support
- define the symbolic constant MT in your source file or on the command line with the /D option

Standard include files declare C run-time library functions as they are implemented in the libraries. If you used the Maximum Optimization (/Ox) or Register Calling Convention (/Gr) option, the compiler assumes that all functions should be called using the register calling convention. The run-time library functions were compiled using either the C or the Pascal calling convention, and the declarations in the standard include files tell the compiler to generate correct external references to these functions.

C Run-Time Library Functions for Thread Control

Any thread can create additional threads. A thread can complete its work very quickly and then terminate, or it can stay active for the life of the program.

LLIBCMT.LIB and LLIBCDLL.LIB, for DLL support, provide two functions for thread creation and termination: the _beginthread and _endthread functions. They also define the global variable _threadid, which contains the address of an application's current thread identifier.

The `_beginthread` function creates a new thread and returns a thread identifier if the operation is successful. The thread will terminate automatically if it completes execution, or it can terminate itself with a call to `_endthread`.

The global variable `_threadid` holds the address of the identifier of the current thread. It is defined in the `STDDEF.H` file as shown:

```
extern int far * _threadid;
```

The `_beginthread` Function

The `_beginthread` function creates a new thread. A thread shares the code and data segments of a task with other threads in the task but has its own unique register values, stack space, and current instruction address. The operating system gives CPU time to each thread, so that all threads in a task can execute concurrently. You can find a complete description of `_beginthread` and its arguments in online Help. Note the following when using the `_beginthread` function:

- The `_beginthread` function lets you pass arguments to the thread
- The stack address points to the bottom of the stack. It is the address of the start of an array or of the start of a block of dynamically allocated memory.
- If you specify `NULL` for the stack address, `_beginthread` manages allocation and deallocation of the thread stack for you. This option is advantageous because it is difficult for your program to determine when a thread has terminated, so you cannot know when to deallocate the thread stack. However, `_beginthread` maintains enough information to know when a thread has terminated and deallocates the thread's stack the next time its thread ID is used.

The `_beginthread` function returns the thread ID number of the new thread if successful or -1 if there was an error. Errors include specifying an odd-address stack or an odd- or zero-length stack (which is different than passing `NULL` for the stack address) or trying to create too many threads.

The `_endthread` Function

The `_endthread` function terminates a thread created by `_beginthread`. Although threads terminate automatically when they complete, the `_endthread` function is useful for conditional termination from within a thread. A thread dedicated to communications processing, for example, can quit if it is unable to get control of the communications port. You can find a complete description of `_endthread` in online Help.

Writing a Multithread Program

When you write a program with multiple threads, you must coordinate their behavior and use of the program's resources. You must also make sure that each thread receives its own stack.

Sharing Common Resources

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, static data, and heap memory, are shared by all threads in the task. Threads using these common resources must coordinate their work.

When multiple threads are accessing static data, your program must provide for possible resource conflicts. Consider a program where one thread updates a static data structure containing x,y coordinates for items to be displayed by another thread. If the update thread alters the x coordinate and is preempted before it can change the y coordinate, the display thread may be scheduled before the y coordinate is updated. The item would be displayed at the wrong location. You can avoid this type of problem by using semaphores to control access to the structure.

Using semaphores is a way of communicating among threads or processes that are executing asynchronously of one another. This communication is usually used to coordinate the activities of multiple threads or processes, typically by controlling access to a shared resource by "locking" and "unlocking" the resource.

To solve the x,y coordinate update problem, the update thread would set a semaphore indicating that the data structure is in use before performing the update. It would then clear the semaphore when both coordinates had been processed. The display thread must wait for the semaphore to be clear before updating the display. This process of waiting for a semaphore is often called blocking on a semaphore because the process is blocked and cannot continue until the semaphore clears.

Screen displays and static data are only two of the resources requiring careful management. For example, your program may have multiple threads accessing the same file. Since another thread may have moved the file pointer, each thread must reset the file pointer before reading or writing.

In addition, each thread must make sure that it is not preempted between the time it positions the pointer and the time it accesses the file. These threads should use a semaphore to coordinate access to the file.

Thread Stacks

You must allocate memory to provide a separate stack for each additional thread your program needs. You must do this before creating the thread. Stack checking, if enabled, is performed for each thread.

Threads that make calls to the C run-time library must allow sufficient stack space for the library functions they call. The C `printf` function, for example, requires more than 500 bytes of stack space. To be safe, allocate at least 4K for each thread's stack.

Since each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. Try to restrict global variables to either RAM semaphores or variables that never change once they are initialized.

Dynamic Linking with CTOS III

A CTOS III dynamic-link library (DLL) is an executable file containing functions that are available to other programs. A statically linked program contains the code and data for all its component functions. In a dynamically linked program, the program-build step does not link all of the code. Instead, CTOS III links calls to functions in dynamic-link libraries at program load time or while the program is running. The DLL code and data become part of the address space of each program, even when the DLL is being accessed by several application programs.

Overview of Dynamic Linking

Dynamic linking is the process of resolving external calls when a program runs, instead of at link time. It offers several benefits:

- Multiple programs can use the same dynamic-link library simultaneously. Since only one copy of the DLL is in memory, there are fewer demands for physical memory and swap space.
- Updates to dynamic-link libraries do not affect the programs that use them, since the only connection between DLLs and application programs is the function-calling sequence.
- Application programs require less disk space and memory, since their executable program files contain the names of DLL functions but not the code for the functions.
- Dynamic-link libraries can call other dynamic-link libraries.

Load-Time and Run-Time Linking

Dynamic linking can take place both at program load time and while the program is running. A program can call functions in more than one DLL and combine both load-time and run-time linking.

The linker creates special records containing the name of each DLL function and the name of its DLL file. It does not put any DLL code into the program's executable file. At load time, CTOS III dynamically links the program and its DLLs. It brings the program and the DLLs into memory and updates the program's DLL calls with the address of each DLL routine. If a DLL is already in memory, it is not reloaded.

With run-time dynamic linking, the program creates the DLL file name and function names during execution. The program then passes these names to CTOS III so the operating system can load the dynamic-link library.

Application Programs and DLLs

With static linking, all library code is bound into the executable program when you link the program. If the library changes, all programs using the library must be relinked.

You can create loosely coupled applications and DLLs and modify the DLLs without relinking the program. For example, if your product has an underlying database access mechanism, you can package the database access routines into a DLL. You can then provide improvements or changes to the database code in a new dynamic-link library. The executable files for the program do not have to be relinked or redistributed.

The programs calling a DLL are known as the DLL's clients.

DLLs and CTOS Microsoft C Run-Time Libraries

You can construct two types of dynamic-link libraries with the CTOS Microsoft C compiler and run-time libraries. Both of them can be multithreaded; they can support more than one client at a time.

The two types of DLLs are:

- A stand-alone dynamic-link library that includes both your routines and code for the CTOS Microsoft C run-time library functions used by your DLL. This type of DLL is self-contained and completely independent of the programs that call it.
- A dynamic-link library that does not use any functions from the CTOS Microsoft C run-time library. This type of DLL is also self-contained.

Standalone Dynamic-Link Libraries

If you want to call C run-time library functions in your DLL, you can include the functions you need. These run-time functions are statically linked in the DLL and the DLL does not rely on the client or any other DLL for run-time support.

Figure 16-1 illustrates the relationships between this type of DLL, an application program, and C run-time library functions. Both the application program and the dynamic-link library have their own copies of functions from the C run-time library. This ensures that:

- The DLL always has access to the C run-time library routines it needs.
- The DLL is not dependent on the calling application for any support code.
- The programs using the DLL do not depend on the DLL for C run-time library functions.

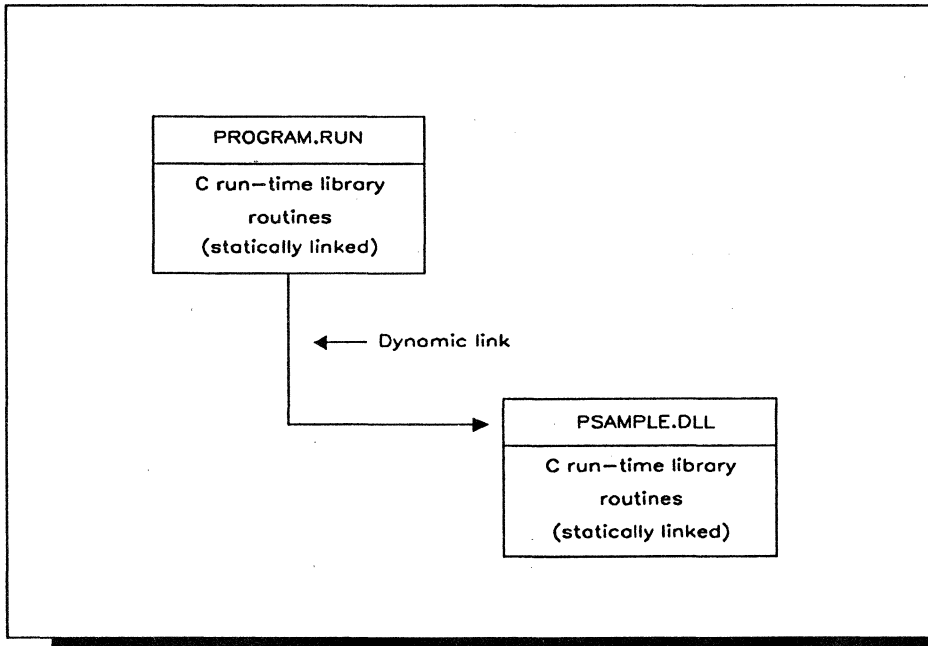


Figure 16-1. DLL and Program with Statically Linked C Run-Time Functions

DLLs without C Run-Time Library Functions

You can write a dynamic-link library in C without calling any functions from the C run-time library. These DLLs contain only your code and require no run-time library support; they make no calls to run-time library functions.

Designing and Writing DLLs

Before you write a DLL, you must determine some of the DLL's requirements. You need to know:

- Floating-point math requirements
- Special initialization requirements such as allocation of buffers or registration of special termination routines
- Termination requirements such as releasing allocated memory
- Re-entrancy requirements

If the DLL is to be called by more than one process, it must be re-entrant.

This subsection explains how to design a DLL to take these requirements into account.

Floating-Point Math Requirements

Stand-alone DLLs built with the LLIBCDLL.LIB library are independent of the programs calling them. They are black boxes that must operate without knowing anything about their client programs and without interfering with their clients.

One area of potential conflict for stand-alone DLLs is control of an 80x87 math coprocessor. For a DLL to use an 80x87 coprocessor or the emulator floating-point library, the DLL and all of its client programs must agree on which process is going to handle floating-point exceptions and on which process is going to handle emulation if the machine does not have a coprocessor.

Floating-point emulation is not possible with a genuinely independent DLL. A stand-alone DLL must use the alternate math library, which ignores the math coprocessor chip. The alternate math library provides the fastest processing available without a coprocessor, but results are not as accurate as those produced by the emulator floating-point library. Because the constraint applies only to the DLL and not to applications, clients of a stand-alone DLL can use any floating-point model. Since the DLL uses the alternate math library, it does not conflict with clients over control of the math coprocessor.

Initialization and Termination Requirements

When you design a DLL, you must decide if it has special initialization or termination requirements. If the DLL needs to initialize variables or allocate memory buffers when it starts, it needs custom start-up procedures. If the DLL acquires system resources for a client program, the resources must be released when the program completes its processing.

Initialization

All DLLs built with `LLIBCDLL.LIB` must use per-process initialization to set up the C run-time data. Per-process initialization (also known as instance initialization) means that CTOS III calls the DLL's initialization code each time it loads a program linked with the DLL. For most DLLs, the default initialization routine is sufficient, and you do not need to take any other measures.

The C run-time library initialization function is called each time a new client is attached to the DLL. To override the default initialization, you must link your DLL with `DLLINIT.OBJ`, the initialization module for DLLs built with `LLIBCDLL.LIB` and using C run-time library code.

In addition, you must declare an entry point for your own DLL initialization function. Your function, or the application program calling your DLL, must initialize the C run-time data by calling the library function `C_INIT` before any other C run-time library functions are called.

The prototype for C_INIT is

```
void _far _pascal C_INIT(void);
```

To have your custom function recognized as the DLL's default initialization routine, it must be the starting point for the DLL. This requires an assembly language file with an END statement naming your function. The sample, SETENTRY.ASM, in the following example shows the minimum assembler code required for specifying a C language function named SampleInit as the DLL's entry point.

```
; SETENTRY.ASM

extrn _SampleInit:FAR           ;name of C start-up routine

end    _SampleInit
```

The following example, SAMPLE.C, shows a simple custom initialization routine that maintains a count of how many clients it is currently serving. Since this example overrides the default dynamic-link library initialization, it must return a nonzero status code to show a successful start-up. If a DLL initialization function returns a status of 0, CTOS III will not load the program that uses the DLL.

```
/* SAMPLE.C */

void _far _pascal C_INIT(void);

int UserCount = 0;

int _export _loadds SampleInit()

{

    UserCount++;           /* increment number of users */

    C_INIT();              /* initialize C run-time data */

    return(1);             /* indicate successful start */

}

/* Code for other DLL functions belongs here.  */
```

All DLLs must be linked with a module-definition file that contains a **LIBRARY** statement, such as the following:

```
LIBRARY SAMPLE INITINSTANCE
```

Refer to the *CTOS Development Utilities Programming Reference Manual* for information on the Module Definition utility.

Note: *For DLLs linked with CTOS Microsoft C run-time libraries, the **LIBRARY** statement in the DLL's module-definition file must specify **INITINSTANCE** in the initialization field. If you omit this, the initialization routine is called only when the DLL is loaded into memory for the first client program, and the DLL will not function properly if it is called by additional programs.*

Termination

You may need to know when an application using your DLL is finished. If your DLL has created buffers or other resources for a particular application, they must be released when the application terminates.

The start-up routine for dynamic-link libraries built with the **LLIBCDLL** library calls specifies a default termination function. To replace the default processing with your own function, link the module **DLLTERM.OBJ** into the DLL. This suppresses the call to **DosExitList**. During initialization, your DLL must register its own routine by calling **DosExitList** unless you are sure the termination routine will be called explicitly. The termination processing must include a call to the library function **C_TERM**.

The prototype for **C_TERM** is:

```
void _far _pascal C_TERM(void);
```

Making the DLL Re-Entrant

Re-entrant code is code that can be shared by multiple programs in a multitasking environment. DLLs that may be used by more than one program must be re-entrant. To do this, they must isolate each client program's data and resources. File handles belonging to one client, for example, must not be used for other clients. Re-entrancy also means that the DLL cannot allow itself to be switched to a different thread while it is performing certain operations.

Global Versus Instance Data

Separate data segments are known as instance data. With instance data segments, the DLL does not have to keep track of which resources belong to each client. CTOS III assigns a different data segment to each process calling the DLL, even though the selectors are the same.

A dynamic-link library can also have a global data segment used for internal purposes or to support all of the programs using its services.

A DLL providing time and date conversions might, for example, keep the current date in a global storage area. The same DLL might provide functions to compute elapsed time, such as the number of minutes between two clock readings. If static variables are used by the elapsed time functions, they should be in instance data segments, since the CTOS III scheduler might preempt the function and schedule another thread that calls the same function with different arguments before it has completed the first caller's task.

Data sharing is controlled by DATA and SEGMENTS statements in a dynamic-link library's module-definition file. By default, a DLL's automatic data segment (the local stack and heap) is shared by all processes calling the DLL. You can specify a unique automatic data segment for each client process by specifying DATA MULTIPLE.

Note: *DLLs built with LLIBCDLL.LIB must use DATA MULTIPLE in the module-definition file.*

Using the SEGMENTS statement allows you to have both global and per-process (instance) data in the same DLL. The C run-time data segment must be per-process. The following is an example of a C program fragment and module-definition file that implement both instance and global data:

```
/* Define static data in the shared segment SHR_SEG. */  
  
int _based(_segname("SHR_SEG")) intvar;  
  
char _based(_segname("SHR_SEG")) charvar;
```

In the module-definition file, define all data segments as nonshareable, then override that default for SHR_SEG as follows:

```
DATA MULTIPLE NONSHARED

SEGMENTS

    SHR_SEG      CLASS    'FAR_DATA'  SHARED
```

Global data segments are created when CTOS III brings the dynamic-link library into memory for its first client process. All of the processes calling the DLL share the same global variables.

Using CTOS Microsoft C Keywords

The `_export` and `_loadds` keywords simplify writing DLLs. They are used to define or declare functions or pointers to functions. In the DLL, an exported function with a single argument might be defined as:

```
int _export _loadds sample(int)
```

The `_export` Keyword

The `_export` keyword gives a function the export attribute. Stack checking must be disabled for exported entry points. You can use the `/Gs` compile option or the `check_stack` pragma to accomplish this.

Using the `_export` keyword is an alternative to declaring the name of the function in the `EXPORTS` section of a module-definition file. It assigns certain default attributes: no I/O privilege, shared data, load on demand, and no alias name. If the defaults are not acceptable, you must specify the proper attributes in the module-definition file.

Not all functions in a DLL are for external use. A DLL can have any number of utility subroutines supporting the work of the exported functions. Functions that are private to the DLL should not have the `_export` keyword.

The `_loadds` Keyword

Upon entry to a DLL, the DS (data segment) register points to the calling program's data segment. To access the DLL's data, the DS register has to be loaded with the DLL's segment selector. The `_loadds` keyword causes the compiler to add prolog and epilog code to the function. The prolog code initializes the DS register to point to the function's data group. The epilog code restores the caller's DS register when the function terminates.

Since loading the DS register is a high overhead operation, you should limit the use of `_loadds` to the exported functions in your DLL.

Note: *Do not use the `_loadds` keyword in a function definition if the function uses only stack variables. If you specify `_loadds` in a DLL that does not have any static data, the linker will issue a segment fix-up error.*

Compile Options for Dynamic-Link Libraries

Dynamic-link libraries must be compiled with specific options that control linking, memory models, and library selection.

Compile without Linking (`/c`)

You must use the `/c` option to build your DLL in separate compile and link steps. This is necessary because the DLL must be linked with a module-definition file specifying that the output file is a dynamic-link library.

Large Memory Model with Separate Stack (`/ALw`)

The `/ALw` option instructs the compiler to use the large memory model with a separate stack segment. Because all DLLs use the caller's stack, you must use `/Aw` or `/Au`. The `/Aw` option sets up separate stack and data segments but does not cause the DS register to be reloaded at the entry to each function. This allows you to call private functions (functions that you do not export) without incurring the overhead of loading the DS register.

Functions that you do export must also be declared using the `_loads` keyword, described above, which sets up the proper DS register handling. If you use the `/Au` option, the DS register will be reloaded on entry to every function, which can cause the function calls in your DLLs to execute more slowly.

All DLL functions are reached using far calls. Pointers passed to and from the DLL must be far pointers.

Remove Stack Probes (/Gs)

Since the DLL uses the caller's stack, you should usually use the `/Gs` option to disable stack checking within the DLL.

Specify 80286 Code (/G2)

Use the `/G2` option to designate code generation for the 80286 processor instruction set.

Link C Run-Time into Stand-Alone DLL (/ML)

Use the `/ML` option to build a stand-alone dynamic-link library that includes static code for C run-time library functions. This option has the same effect as using the `/ALw`, `/FPa`, `/G2`, and `/D MT` options. See *DLLs with Static C Run-Time Library Functions* for more information about these options.

Link Executable or DLL with C Run-Time DLL (/MD)

Use the `/MD` option to build an executable file or a dynamic-link library that calls a C run-time DLL. This option has the same effect as using the `/ALw`, `/FPi`, `/G2`, `/D DLL`, and `/D MT` options. See *Programs and DLLs with a C Run-Time DLL* for more information about these options.

Building DLLs with CTOS Microsoft C

To build a DLL, compile and link the dynamic-link library like any other version 8 run file, but add a module definition file. This module definition file tells the linker that the output is a dynamic-link library.

When you build applications that use a dynamic-link library, you must tell the linker where to find the library's dynamically linked functions. You use import libraries and module-definition files for this purpose.

DLLs with Static C Run-Time Library Functions

LLIBCDLL.LIB is used to create stand-alone DLLs. The library functions are re-entrant and can be called by multiple threads within a program as well as by multiple programs. The code for the stand-alone DLL's C run-time library functions is contained within the DLL. Programs that call stand-alone DLLs have their own run-time library

Building the DLL

The files required to build a stand-alone DLL with LLIBCDLL.LIB are listed in the following table.

Table 16-2. Stand-Alone DLL Files

File Name	Description
SystemImp.Lib	CTOS III import library for system.dll
LLIBCDLL.LIB	Large-model multithread C run-time library for DLLs
DLLINIT.OBJ	Optional initialization module for DLLs requiring custom initialization
DLLTERM.OBJ	Optional termination module for DLLs requiring custom exit processing
<u>userdll.C</u>	Source code for the DLL you create
<u>userdll.DEF</u>	Module-definition file for the DLL you create

The module JUSTIFY.C is an example of source code for a simple dynamic-link library. The RightJustify routine calls the strlen function from the C run-time library and right-justifies a caller's buffer. The function definition includes the `_export` keyword. The `_loads` keyword is omitted, since this function does not need any static data. If it did, you would need to specify `_loads`.

For simplicity, JUSTIFY.C shows a DLL with a single function. In actual practice, you would usually package a group of similar utilities into one DLL.

```
/* JUSTIFY.C -- Sample Dynamic-Link Library */

#include <string.h>

/* Right justifies the string in TargetBuff to TargetSize
 * and inserts necessary number of FillChars on the left.
 */

#pragma stack_check(off)

int _export RightJustify(char *TargetBuff, int TargetSize,
                        char FillChar)

{
```

```
char *s, *d;

s = TargetBuff + strlen(TargetBuff);

d = TargetBuff + TargetSize;

while (s = TargetBuff)

    *d-- = *s--;

while (d = TargetBuff)

    *d-- = FillChar;

return(0);

}
```

To create a stand-alone dynamic-link library with JUSTIFY.C, follow this procedure (the DLL in the example is named JUSTLIB1.DLL):

1. Compile with the /ML Option.

Compile the source file without linking. Dynamic-link libraries linked with LLIBCDLL must be compiled with specific options.

Use the /ML option to automatically select the following options:

Option	Effect
/ALw	Use large memory model with separate stack segment
/G2	Use 80286 processor instruction set
/D MT	Use the multithread definitions in the include files
/FPa	Generate floating-point calls and select the alternate math library

The /G2 and the /ALw options can be overridden.

You should also use the /Gs option to suppress stack checking and the /c option to compile without linking. The complete command to compile the sample file JUSTIFY.C is:

```
CL
  [Args]  /ML /Gs /c Justify.C
  [      ]
```

2. Create a module-definition file.

Create a module-definition file, JUSTLIB1.DEF, which includes the following lines: LIBRARY JUSTLIB1 INITINSTANCE DATA MULTIPLE

The LIBRARY statement identifies the executable file, JUSTLIB1.DLL, as a dynamic-link library. DLLs linked with the LLIBCDLL library must specify INITINSTANCE in the initialization field. You could add an EXPORTS statement for the RightJustify function in JUSTIFY.C, but it is optional since the _export keyword was used in the source code.

3. Link with LLIBCDLL.LIB.

Ensure that the file LLIBCDLL.LIB, which takes the place of the regular C run-time library, is available.

Note: When you link with LLIBCDLL.LIB, you cannot have any other C run-time libraries in the link.

4. Create an import library.

Applications that call DLLs use import libraries to identify DLL functions to the linker. The following example uses JUSTLIB1.DEF and the Module Definition utility to create an import library named JUSTLIB1.LIB:

```
Module Definition
[Input File(.DEF)]  JUSTLIB1.DEF
[Object module]
[Import library]    JUSTLIB1.LIB
```

Building Programs that Call the DLL

To link a dynamic-link library with an application, you must have one of the following:

- A module-definition file with an IMPORTS statement for each DLL function called by your program
- An import library created from the DLL itself or from a module-definition file

All calls to DLLs must be far calls; all pointers passed must be far data pointers. If you do not compile with the large memory model option (/AL), you must cast the DLL function calls and pointers yourself.

The sample file TESTJUST.C is compiled and linked into a small-model program named SAMPLE1.RUN. TESTJUST.C includes a function prototype that declares RightJustify as a far function expecting a far pointer as its first argument. Because of the prototype, the compiler will generate a far call to RightJustify and coerce the pointer argument to the proper value.

```
/* TESTJUST.C. Call sample DLL library */

#include <stdio.h>

#include <string.h>

/* DLL function prototype */

int _far RightJustify(char _far *, int, char);

void main(void)

{
```

```
char buff[12];

strcpy(buff, "ABCD");

/* Right justify to 8 characters and zero fill. */

RightJustify(buff, 8, '0');

printf("Result: %s\n", buff);

}
```

You need several files to link an application with a stand-alone DLL. The following table lists them.

Table 16-3. Linking Files Needed

File Name	Description
<u>userdll</u> .LIB	Import library file for the DLL
<u>userapp</u> .DEF	Optional module-definition file for your application that contains an IMPORTS statement for each DLL function called (required if not using an import library)
SystemImp.Lib	Import library file for CTOS III's system.dll (required if your application uses the C library)
userapp.OBJ	Object module(s) for your application

Additional Information

For additional information on dynamic-link libraries and the module definition utility, refer to the following:

- Dynamic-link libraries--*CTOS Operating System Concepts Manual*
- Module Definition utility--*CTOS Programming Utilities*

This Page is Blank
Do Not Print

Appendix A

Migrating to CTOS Microsoft C

Migrating from BTOS C to CTOS Microsoft C is relatively straightforward because BTOS C conforms very well to ANSI C. BTOS C programs developed with portability in mind are fairly easy to convert. In many cases, the conversion process is as simple as recompiling with new header files, and re-linking with new libraries. Nevertheless, converting programs that deviate from good programming practice (developed without portability in mind) may take longer.

This appendix describes the following migration considerations:

- Program Changes
- Compiler Options
- Predefined Macros
- COMDEF OMF versus PUBDEF Records
- DS Allocation
- Overlays
- C Library Functions
- Math Library Functions

Section 5, Building Applications, also contains information that relates to migration issues, especially under the heading Programming Notes.

Program Changes

To migrate from BTOS C to CTOS Microsoft C, you may have to change your program source code. Change considerations exist in the following areas:

- C functions
- Header files
- Memory models
- Language keywords
- Pragmas
- Mixed-language programming
- Coding Conventions

C Functions

The BTOS C functions documented in the *BTOS C Compiler Programming Reference Manual* are nearly a subset of the CTOS Microsoft C functions. Programs that use documented functions only, may require minor changes only. Programs that use the few undocumented (internally used) BTOS C functions require more work, because CTOS Microsoft C does not support the undocumented functions.

The three tables that follow list documented and undocumented BTOS C functions, show the support provided by CTOS Microsoft C, and list substitute functions if they exist. As such, these functions fall into three categories:

- Documented BTOS C Functions Without Functional Substitutes
- Documented BTOS C Functions With Functional Substitutes
- Undocumented BTOS C Functions With Possible Work-arounds

Following the tables, you will find information about three functions that affect:

- Time
- Error Codes
- Input/Output

For a complete list of BTOS C functions, libraries, and associated CTOS Microsoft C support, refer to the headings C Library Functions and Math Library Functions, described later in this section. The tables shown here provide summaries only.

CTOS Microsoft C does not support the documented BTOS C functions listed in table A-1. You must recode your program to replace the BTOS C functionality. For reference, refer to the library source code provided with BTOS C.

Table A-1. Documented BTOS C Functions Without Substitutes

gsignal	peekb	ringdex	stime	vsscanf
index	poke	ssignal	vfscanf	
peek	pokeb	ssort	vscanf	

Table A-2 lists documented BTOS C functions that have functional substitutes in CTOS Microsoft C. Simply replace the BTOS C function name with the new CTOS Microsoft C function name.

Table A-2. Documented BTOS C Functions With Substitutes

BTOS C Function	CTOS Microsoft C Function
inport	inpw
inportb	inp
memmove	movmem
outport	outpw
outportb	outp
setmem	memset

Table A-3 lists undocumented BTOS C functions, none of which CTOS Microsoft C supports. If these functions appear in your programs, you must recode the functionality with a work-around. One example of an undocumented function is `fdtofh`, which converts a C file handle to a CTOS file handle. For details, refer to the library source code provided with BTOS C.

Table A-3. Undocumented BTOS C Functions With Work-arounds

BTOS C Function	For Possible Use, Refer to CTOS Microsoft C's:
check8087	---
coreleft	_freect, _memavail, _memmax
coreminleft	_freect, _memavail, _memmax
fdtofh	---
myds	FP_SEG
truncfile	chsize

CTOS Microsoft C considers `time_t` a long integer, BTOS C considers it a structure. Under CTOS Microsoft C, `time_t` returns the number of seconds elapsed since January 1, 1970, 00:00:00 GMT (Greenwich mean time). Under BTOS C, `time_t` is a structure that represents time as returned by `GetDateTime`, which is the CTOS operating system format. Therefore, depending on how your code manipulates the value returned, you may need to recode computations and manipulations.

CTOS Microsoft C holds error codes returned from CTOS calls with the variable `_ctoserrno`. BTOS C holds these error codes in the global variable `errno`.

If your program uses a CTOS file handle returned by `fdtofh` to call the CTOS function `ChangeFileLength`, substitute the CTOS Microsoft C `chsize` function to either truncate or extend a file. The `chsize` function also substitutes for the BTOS C function `truncfile`.

Header Files

Header files (.h) used by BTOS C and CTOS Microsoft C differ somewhat in these areas:

- `#include` file names
- `#defines`
- `typedefs`
- structure definitions

Differences in these areas are divided into the following topics:

- Notes
- Using `ctos.lib.h`
- Header File Conversions

Notes

You will have to replace many header file names in `#include` statements. For example, you must replace all references to `ctos.h` with `ctos.lib.h`.

Values associated with `#defines` may differ between BTOS C and CTOS Microsoft C header files. However, if `#define` values are referenced explicitly by name, problems are unlikely.

Some BTOS C header files contain `#defines`, `typedefs`, and structures not found in CTOS Microsoft C header files. For example, `ctos.lib.h` contains no structure definitions. To create compatibility, try building a new header file that contains the missing material, such as `old.btosc.h`. Sometimes, however, you must use CTOS Microsoft C work-around logic.

Using ctos.lib.h

To use `ctos.lib.h`, you must insert `#define` statements for each CTOS.Lib procedure used in the module before the `#include <ctos.lib.h>` statement.

Also, it is crucial to maintain correct upper- and lower-case format in `#define` statement identifiers. If the case is incorrect, no error messages appear when compiling unless you use switches `/W3` or `/W4`, which generate warning messages when unprototyped procedures are called.

Another way to find identifiers with incorrect case is to append an illegal C character to `#define` statements, as shown in this example:

```
#define AllocExch      @
#define ErrorExitString @
#include <ctos.lib.h>
```

For `#define` statements with correct case, the header file undefines the identifier, and establishes it as a procedure. For `#define` statements with incorrect case, `@` will be substituted for the function identifiers, causing compiler errors that point to the problem statements.

Header File Conversions

Table A-4 lists BTOS C header files along with the CTOS Microsoft C counterparts. Differences between the two appear in the CTOS Microsoft C column. The word Same indicates identical header file names. Again, note that `ctos.lib.h` contains no structure definitions.

Table A-4. BTOS C and CTOS Microsoft C Header Files

BTOS C Header File	CTOS Microsoft C Header File
assert.h	Same
ctclust.h	ctos.lib.h
ctcomm.h	ctos.lib.h
ctcont.h	ctos.lib.h
ctdam.h	ctos.lib.h
ctexch.h	ctos.lib.h
ctfile.h	ctos.lib.h
ctinter.h	ctos.lib.h
ctkeybd.h	ctos.lib.h
ctmem.h	ctos.lib.h
ctmsg.h	ctos.lib.h
ctos.h	ctos.lib.h
ctparm.h	ctos.lib.h
ctpart.h	ctos.lib.h
ctproc.h	ctos.lib.h

continued

Table A-4. BTOS C and CTOS Microsoft C Header Files (cont.)

BTOS C Header File	CTOS Microsoft C Header File
ctqueue.h	ctos.lib.h
ctrsam.h	ctos.lib.h
ctserv.h	ctos.lib.h
ctspool.h	ctos.lib.h
ctstam.h	ctos.lib.h
ctstream.h	ctos.lib.h
cttask.h	ctos.lib.h
cttimer.h	ctos.lib.h
ctvideo.h	ctos.lib.h
ctvirt.h	ctos.lib.h
ctype.h	Same
ermo.h	Same; some of the #defines have different values
float.h	Same; some of the #defines have different values
i8086.h	ctos.lib.h, dos.h (see XREG, HREG, SREGS); some of the structure definitions and #defines are not available
limits.h	Same; some of the #defines have different values

continued

Table A-4. BTOS C and CTOS Microsoft C Header Files (cont.)

BTOS C Header File	CTOS Microsoft C Header File
math.h	Same
setjmp.h	Same
signal.h	Same
stdarg.h	Same
stddef.h	Same
stdio.h	Same; some structures (for instance, FILE) are different and some of the #defines have different values
stdlib.h	Same
string.h	Same
time.h	Same

Memory Models

Both CTOS Microsoft C and BTOS C support the same implementation for Small, Medium, and Large memory models. They do not, however, support the same implementation for the Huge memory model. Thus, migration becomes an issue when handling the Huge memory model, and two methods exist for converting Huge models from BTOS C to CTOS Microsoft C. Both methods involve special handling of the CTOS Microsoft C Large memory model.

To understand how the methods work, however, it helps to understand how the two compilers implement the Huge model. BTOS C maintains separate Data segments for each module; CTOS Microsoft C does not. So unless initialized data exceeds 32,767 bytes in length, the CTOS Microsoft C compiler assigns it to DGroup, but nothing stops the Huge model program from exceeding DGroup space. So the CTOS Microsoft C approach is to add Huge pointer capability to its Large model. Huge pointers are far pointers to data items that can exceed 64K.

To migrate between implementations, both methods use compiler switches. Method one uses compiler switches to move initialized data to Data segments outside DGroup. Method two uses compiler switches to customize the CTOS Microsoft C Large memory model. This customization mimics the separate-Data-segment-per-module attribute used by the BTOS C Huge memory model.

To use method one, specify the compiler options `/AL` and `/Gt[number]`. The `/AL` option compiles with the Large memory model. The `/Gt[number]` option moves data items greater than or equal to `[number]` bytes outside DGroup to a new segment. To collect items relocated with this option, the compiler assigns data segments as needed. To relocate uninitialized data, simply initialize the first element, which causes the data to become initialized, and the `/Gt` option processes it accordingly.

To use method two, specify the compiler options `/AL`, `Au`, and `/ND`. The `/AL` option compiles with the Large memory model. The `/Au` option generates additional function-entry and -exit code. When entering a function, the system saves the DS register value, and loads the DS value for the module that contains the function definition. Upon function exit, DS is restored. The `/NDdataseg` option uniquely names each module's Data segment. For example, to produce the same data segment name as BTOS C's huge model, specify `"D_ "<module name>`.

The functions in the Microsoft C run-time libraries that rely on the default data segment being named `_DATA` will fail if you use the `/ND` option to rename the default data segment. You must use the `/Gu` option with `/ND` to reset the DS register to the `_DATA` segment before calling C-library functions.

The `/ND` option (method two) is intended for use with C source modules that contain data only. For modules that contain code, the `/Gt` option (method one) is preferred, because of the high run-time overhead of reloading the DS register.

Language Keywords

Most BTOS C keywords are Standard C, and work the same way in CTOS Microsoft C. However, some keywords require a name change. Other keywords require a name change along with a position change in declarations and statements. In addition, all CTOS Microsoft C extended keywords begin with an underscore (`_`) character.

The compiler recognizes the non-underscore versions of these keywords, but they do not conform to ANSI C, and using them is considered obsolete.

Table A-5 shows BTOS C extended keywords and the corresponding CTOS Microsoft C keywords. Changes in extended-keyword usage are also noted. Standard C keywords (such as `if` and `while`) do not appear because they are the same for both BTOS C and CTOS.

Table A-5. BTOS C and CTOS Microsoft C Extended Keywords

BTOS C Extended Keyword	CTOS Microsoft C Extended Keyword (and Usage Changes)
asm	_asm, with possible different opcode usage and syntax
far	_far, with different positioning in function prototype or when used with * pointer token
interrupt	_interrupt, with different positioning in function prototype or statement
near	_near, with different positioning in function prototype or when used with ** pointer token
plm	_plm For usage details, refer to Section 5, Building Applications, under the heading Programming Notes.
_cs, _ds, _es, _ss	Use the FP_SEG macro in <dos.h> or in-line assembler to retrieve register value

The following declarations of pointer to an integer show the ordering difference of near and far keywords, of the * pointer token:

```
BTOS C:  int * near pi;
CTOS Microsoft C:  int _near * pi;
```

Pragmas

No pragmas are defined for BTOS C, so pragmas are not a migration issue.

Mixed-Language Programming

Like BTOS C, CTOS Microsoft C supports mixed-language code. Be aware, however, that names for Code and Data segments differ between BTOS and CTOS Microsoft C. Segment classes for Data segments also differ.

Coding Conventions

Coding-convention differences exist in these areas:

- Function Prototypes
- Initialization
- Function and Macro Names

Function Prototypes

CTOS Microsoft C completely prototypes functions in header files, BTOS C does not. For example, both implementations contain the header file `STRING.H`, and both files contain the heavily-used function `strcmp()`. However, `strcmp()` appears differently in the header files.

BTOS C

```
int strcmp();
```

CTOS Microsoft C

```
extern int _FAR _cdecl strcmp(const char _FAR *,  
                             const char _FAR *);
```

As a result, CTOS Microsoft C assumes that received parameters conform to strict prototypes. So if you get compiler warnings in this area, you may have to cast the arguments to conform to the prototypes.

Initialization

BTOS C allows initialization of variables without `=`. For example, the following syntax in BTOS C successfully initializes an integer and a structure:

```
int i 0;

struct point
{ int x;
  int y;
};

struct point pt {2, 4};
```

CTOS Microsoft C requires `=` for initialization, as shown here:

```
int i =0;
struct point pt ={2, 4};
```

Function and Macro Names

BTOS C allows function and macro names to exceed 31 characters. CTOS Microsoft C truncates names longer than 31 characters.

Compiler Options

CTOS Microsoft C does not support some BTOS C compiler options. You may need to recode programs designed to use these options.

Table A-6 identifies CTOS Microsoft C compiler options which may be used in place of the BTOS C compiler options. None means there is no equivalent option.

Table A-6. BTOS C and CTOS Microsoft C Compiler Options

BTOS C Compiler Option	CTOS Microsoft C Compiler Option	Description
-A	/Za	Compile with no BTOS extensions
-C	None	Allow nested comments
-D <i>identifier</i>	/D <i>identifier</i>	Defines <i>identifier</i> to '1'
-D <i>iden</i> = <i>string</i>	/D <i>identifier</i> [= <i>value</i>]	Defines <i>iden(tifier)</i> to <i>string</i>
-E	None	Allow elastic type matches
-G	/Ot (default)	Generate code for speed rather than size
-I <i>directory</i>	/I <i>directory</i>	Specify include <i>directory</i>
-K	/J	Treat char type as unsigned
-L	/W4	Do a lint cross check
-L <i>filename</i>	/F <i>sfilename</i> and /W4	Do a lint compile to lint file <i>filename</i>
-N	/Ge (default)	Generate stack overflow logic
-O	Default	Compile with optimizer
-P	/P, /Ep, or /E	Preprocess source file only
-Q	None	Place only definitions in lint files. Also, consider using /Zg for this function.

continued

Table A-6. BTOS C and CTOS Microsoft C Compiler Options (cont.)

BTOS C Compiler Option	CTOS Microsoft C Compiler Option	Description
-S	/Fa[filename]	Compile to assembly source
-T	None	Warn about explicit casts
-U <i>identifier</i>	/U <i>identifier</i>	Undefine <i>identifier</i>
-Y	Default	Generate full function entry and exit code
-Z	Default	Suppress redundant register loads
-a	/Zp{2 \w 4}	Force word alignment of integers (/Zp only affects structures)
-b	None	Do not warn about unreachable breaks
-d	/W0	Suppress long to int conversion warnings
-f	/Fpi87	Generate in-line 8087 instructions
-g	None	Make all code and data segments PUBLIC, allowing segments with the same name to share the same segment/selector
-h	None	Suppress heuristic test warnings
-i#	/H <i>number</i>	Set identifier length

continued

Table A-6. BTOS C and CTOS Microsoft C Compiler Options (cont.)

BTOS C Compiler Option	CTOS Microsoft C Compiler Option	Description
-mhf	(See Memory Models)	Generate Huge model
-mlf	/AL	Generate Large model
-mmf	/AM	Generate Medium model
-msf	/AS	Generate Small model, 16-bit pointer arithmetic
-n1path	Control with TMP environment variable	Place CC0 output in named path
-n2path	Control with TMP environment variable	Place CC1 output in named path
-nopath	Control with TMP environment variable	Place CC2 output in named path
-p	/Gc with /Zc	Generate PL/M calling sequence (or use PL/M keyword)
-q	None	Suppress undefined symbols in lint exectutions
-r	/r	Suppress register variables
-s	None	Warn about passing structures as arguments

continued

Table A-6. BTOS C and CTOS Microsoft C Compiler Options (cont.)

BTOS C Compiler Option	CTOS Microsoft C Compiler Option	Description
-w	/w or /W0	Suppress all warnings
-wxxx	/Wn	Suppress warnings
-x	None	Warn about unused externs
-y	/Zd	Generate source line numbers in object code
-zAname	None	Set Code segment class
-zBname	None	Set Data segment class
-zCname	/NTname	Set Code segment name
-zDname	/NDname	Set Data segment name
-zGname	None	Set Data group name
-zPname	None	Set Code group name
-1	/G1	Generate 80186 instructions
-2	/G2	Generate 80286 instructions

CTOS Microsoft C does not support nested comments.

CTOS Microsoft C defaults to word alignment of structure members (/Zp2), equivalent to the BTOS C -a option. To correctly access CTOS system structures, specify /Zp1.

CTOS Microsoft C does not allow explicit names for Code segment class (-zAname), uninitialized Data segment (-zDname), Data group (-zGname) and Code group (-zPname) as does BTOS C. You may have to redesign programs that depend on this capability.

Predefined Macros

The BTOS C compiler predefines these macros:

<code>__BTOSC__</code>	<code>__FILE__</code>
<code>__LINE__</code>	<code>__DATE__</code>
<code>__TIME__</code>	<code>__SMALL__</code>
<code>__MEDIUM__</code>	<code>__LARGE__</code>
<code>__HUGE__</code>	

The CTOS Microsoft C compiler predefines these macros:

<code>__CTOS__</code>
<code>__TIME__</code>
<code>__DATE__</code>
<code>__TIMESTAMP__</code>
<code>__LINE__</code>
<code>__FILE__</code>

Note: For a complete list, refer to the online Help.

COMDEF OMF Versus PUBDEF Records

CTOS Microsoft C emits COMDEF OMF (COMmunal DEFinition, Object Module Format) records to object files for uninitialized global variables, rather than emit the PUBDEF (PUBLIC DEFinition) records created by BTOS C. Therefore, COMDEF records can cause unexpected behavior in the following ways:

- Multiple source modules may contain identical declarations for the same global variable without causing an error. No extern declaration is required. If the Linker does not discover a PUBDEF in a received object stream, the Linker creates storage for the variable.
- `__far` communal data is not placed in DGroup, the default data segment, so the selection of a COMDEF variable is not equal to the value of the DS or SS registers.

- COMDEFs are invisible to CTOS Librarian and to MLIB, the CTOS Microsoft Librarian. Thus, if you recompile a module with CTOS Microsoft C that the Linker extracted from a library, because it contained an uninitialized data PUBDEF that satisfied an unresolved external reference, the Microsoft C Linker will not find the module.

Therefore, you must specify the desired object module in the link list using the `My.Lib(module_name)` syntax. Alternatively, to initialize the variable, change the data declaration in the module.

- Only PUBDEF identifiers appear in the MLIB and CTOS Librarian cross reference listings.

DS Allocation

When linking a CTOS Microsoft C program, always set the option [DS Allocation] to No. DS Allocation is incompatible with CTOS Microsoft C's DGroup usage.

Overlays

To produce overlay-compatible code, code that conforms the CTOS Virtual Code Segment Management facility requirements enforced by the CTOS Linker, compile with these switches:

`/AL /GY`

For details, refer to Section 5, Building Applications, under the heading Using the Compiler. This section describes these options in detail, and also describes incompatibilities you should be aware of.

C Library Functions

Table A-7 lists all BTOS C library functions (except helper routines) and associated CTOS Microsoft C header files . The function name appears in the left-hand column; status and location appears in the right-hand column. The right-hand column also shows the support status offered by CTOS Microsoft C. The four status possibilities are:

- Supported, where a header file (.h) contains function support. You must include the indicated header files in the program to use the associated function prototype.
- Undocumented, where the function does not appear in the *BTOS C Compiler Programming Reference Manual*, and is not supported by CTOS Microsoft C.
- Substitute, where a CTOS Microsoft C function provides the same functionality.
- Not Supported, where no equivalent function exists in CTOS Microsoft C.

Table A-7. BTOS C Function Status in CTOS Microsoft C

Function Name	Status/ Prototype Location
abs	STDLIB.H, ACCESS.H
asctime	TIME.H
atof	MATH.LIB, STDLIB.H
atoi	STDLIB.H
atol	STDLIB.H

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
bsearch	STDLIB.H, SEARCH.H
calloc	STDLIB.H, MALLOC.H
check8087	Undocumented
close	IO.H, ERRNO.H
coreleft	Undocumented
coreminleft	Undocumented
creat	SYS\TYPES.H, SYS\STAT.H, IO.H, ERRNO.H
ctime	TIME.H
ecvt	STDLIB.H
exit	PROCESS.H, STDLIB.H
fclose	STDIO.H
fcvt	STDLIB.H
fdtofh	Undocumented
fflush	STDIO.H
fgetc	STDIO.H
fgets	STDIO.H

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
fopen	STDIO.H
fprintf	STDIO.H
fputc	STDIO.H
fputs	STDIO.H
fread	STDIO.H
free	STDLIB.H, MALLOC.H
freopen	STDIO.H
fscanf	STDIO.H
fseek	STDIO.H
ftell	STDIO.H
fwrite	STDIO.H
gcvt	STDLIB.H
gets	STDIO.H
getw	STDIO.H
gmtime	TIME.H
gsignal	Not Supported

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
index	Not Supported
init8087	Undocumented
inport	Substitute inpw
inportb	Substitute inp
ldiv	STDLIB.H
localtime	TIME.H
longjmp	SETJMP.H
lsearch	SEARCH.H
lseek	IO.H, STDIO.H, ERRNO.H
malloc	STDLIB.H, MALLOC.H
memchr	STRING.H, MEMORY.H
memcmp	STRING.H, MEMORY.H
memcpy	STRING.H, MEMORY.H
memset	STRING.H, MEMORY.H
movmem	Substitute memmove
myds	Undocumented

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
open	SYS\TYPES.H, SYS\STAT.H, IO.H, ERRNO.H, FCNTL.H
outport	Substitute outpw
outportb	Substitute outp
peek	Not Supported
peekb	Not Supported
poke	Not Supported
pokeb	Not Supported
printf	STDIO.H
puts	STDIO.H
putw	STDIO.H
qsort	STSLIB.H, SEARCH.H
rand	STDLIB.H
read	IO.H, ERRNO.H
realloc	STDLIB.H or MALLOC.H
rindex	Not Supported
scanf	STDIO.H

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
segread	DOS.H
setbuf	STDIO.H
setjmp	SETJMP.H
setmem	Substitute memset
setvbuf	STDIO.H
sprintf	STDIO.H
srand	STDLIB.H
sscanf	STDIO.H
ssignal	Not Supported
ssort	Not Supported
stime	Not Supported
strcat	STRING.H
strchr	STRING.H
strcmp	STRING.H
strcpy	STRING.H
strcspn	STRING.H
strlen	STRING.H

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
strncat	STRING.H
strncmp	STRING.H
strncpy	STRING.H
strpbrk	STRING.H
strchr	STRING.H
strspn	STRING.H
strtod	STDLIB.H, ERRNO.H
strtok	STRING.H
strtol	STDLIB.H, ERRNO.H
swab	STDLIB.H
time	TIME.H
tolower	CTYPE.H
toupper	CTYPE.H
truncfile	Undocumented
ungetc	STDIO.H
unlink	IO.H or STDIO.H, ERRNO.H
vfprintf	STDARG.H or VARARGS.H, STDIO.H

continued

Table A-7. BTOS C Function Status in CTOS Microsoft C (cont.)

Function Name	Status/ Prototype Location
vfscanf	Not Supported
vprintf	STDARG.H or VARARGS.H, STDIO.H
vscanf	Not Supported
vsprintf	STDARG.H or VARARGS.H, STDIO.H
vsscanf	Not Supported
write	IO.H, ERRNO.H
_exit	PROCESS.H, STDLIB.H

Note: CTOS Microsoft C provides both functions and macros for the following functions:

isalnum iscntrl islower isspace

isalpha isdigit isprint isupper

isascii isgraph ispunct isxdigit

Math Library Functions

Table A-8 lists all functions (except C language helper routines) found in the BTOS C Math library, along with the associated CTOS Microsoft C header files that contain the function prototypes.

Table A-8. BTOS C Math Function Locations in CTOS Microsoft C

Function Name	CTOS Microsoft C Header File
acos	MATH.H, ERRNO.H
asin	MATH.H, ERRNO.H
atan	MATH.H, ERRNO.H
atan2	MATH.H, ERRNO.H
ceil	MATH.H
cos	MATH.H, ERRNO.H
cosh	MATH.H, ERRNO.H
exp	MATH.H, ERRNO.H
fabs	MATH.H
floor	MATH.H
fmod	MATH.H
frexp	MATH.H
ldexp	MATH.H, ERRNO.H

continued

Table A-8. BTOS C Math Function Locations in CTOS Microsoft C (cont.)

Function Name	CTOS Microsoft C Header File
log10	MATH.H, ERRNO.H
log	MATH.H, ERRNO.H
modf	MATH.H
pow	MATH.H, ERRNO.H
sin	MATH.H, ERRNO.H
sinh	MATH.H, ERRNO.H
sqrt	MATH.H, ERRNO.H
tan	MATH.H, ERRNO.H
tanh	MATH.H

This Page is Blank
Do Not Print

Appendix B

Command Reference

This appendix shows the command-line options available for the following utilities:

- Compiler
- The CTOS Microsoft CodeView Debugger
- Environment Service
- HelpMake
- MLib
- MLink
- NMake
- Programmer's Workbench
- PWBRMake
- QuickHelp

CL

Summary The CL utility compiles one or more C source files and then invokes MLINK to link the files. The following table lists and briefly describes the options available.

Syntax CL
 `[options] [filename]...[libraries link-options]`

Environment Variables Recognized

CL
TMP
INCLUDE

Command Line Options

Option	Description
<i>/A {SIMICLIH}</i>	Selects one of these standard memory models: <i>/AS</i> Small memory model. Code and data are limited to 64K each. (Same as <i>/Asnd</i>.) <i>/AM</i> Medium memory model. Data is limited to 64K. (Same as <i>/Alnd</i>.) <i>/AC</i> Compact memory model. Code is limited to 64K. (Same as <i>/Asfd</i>.) <i>/AL</i> Large memory model. No limits on code or data. (Same as <i>/Alfd</i>.) <i>/AH</i> Huge memory model. Same as large model, but individual arrays can exceed 64K. (Same as <i>/Alhd</i>.)
<i>/Astring</i>	Sets up a customized memory model. The string consists of three characters in any order, indicating code and data pointer size and segment setup.

Option	Description		
	Group	Code	Description
	Code Pointers	s	Small (Near)
		l	Large (Far)
	Data Pointers	n	Near
		f	Far
		h	Huge
	Segment Setup	d	SS == DS
		u	SS != DS; DS loaded for each function entry
		w	SS != DS; DS not loaded at function entry
	/C	Preserves comments when preprocessing a file; use only with /E, /P, or /EP.	
/c	Compiles without linking. Creates an object file but does not successor call MLINK.		
/D <i>id</i> [= [<i>value</i>]]	Defines the symbolic constant <i>id</i> to the preprocessor. If <i>value</i> is defined, the value of <i>id</i> is <i>value</i> . If the equal sign is given without <i>value</i> , the value of <i>id</i> is empty. If <i>id</i> is given without the equal sign, the value of <i>id</i> is 1.		
/E	Preprocesses the source file, copying the result to the standard output and inserting #line directives.		
/EP	Preprocesses the source file, copying the result to the standard output without #line directives.		
/F <i>hexnum</i>	Sets stack size to <i>hexnum</i> bytes (this is the same as /link /STACK: <i>number</i>). The value must be expressed in hexadecimal notation.		
/Fa [<i>listfile</i>]	Produces an assembly listing. If <i>listfile</i> is unspecified, /Fa defaults to <i>sourcefilename</i> .Asm.		

Command Reference

Option	Description
/Fc [<i>listfile</i>]	Produces a combined source-assembly code listing. If <i>listfile</i> is unspecified, /Fc defaults to <i>sourcefilename</i> .Cod.
/Fe <i>exefile</i>	Names the executable file.
/FI [<i>listfile</i>]	Generates an object-code listing. If <i>listfile</i> is not listed, /FI defaults to <i>sourcefilename</i> .Cod.
/Fm [<i>mapfile</i>]	Creates a linker map file. If <i>mapfile</i> is not given, /Fm defaults to <i>first-sourcefilename</i> .Map.
/Fo <i>objfile</i>	Names the object file. If a path is specified only, such as /Fo[<i>vol</i>]< <i>dir</i> >, the path is pre-pended to the default object file name.
/FPa	Generates floating-point calls. Must be linked with the alternate math library, xLIBFA.LIB. (x = S, M, C, or L).
/FPc	Generates floating-point calls and requires the emulator, EM.LIB, which uses an 80x87 coprocessor if one is present.
/FPc87	Generates floating-point calls and must be linked with 87.LIB, which requires an 80x87 coprocessor at run time.
/FPi	Generates in-line 80x87 instructions and requires the emulator, EM.LIB, which uses an 80x87 processor if one is present. This is the default /FP option.
/FPi87	Generates in-line 80x87 instructions and must be linked with 87.LIB, which requires an 80x87 coprocessor at run time.
/Fr [<i>browsefile</i>]	Generates a standard PWB Source Browser database. If <i>browsefile</i> is unspecified, /Fr defaults to <i>basename</i> .SBR.
/FR [<i>browsefile</i>]	Generates an extended Source Browser database. If <i>browsefile</i> is unspecified, /FR defaults to <i>basename</i> .SBR.
/Fs [<i>listfile</i>]	Produces a source listing. If <i>listfile</i> is unspecified, /Fs defaults to <i>sourcefilename</i> .LST.
/Fx [<i>xreffile</i>]	Specifies a name for the Microsoft Macro Assembler (MASM) cross-reference file. If <i>xreffile</i> is unspecified, /Fx defaults to <i>sourcefilename</i> .CRF.

Option	Description
/G0	Generates 8086/8088 instructions. This is the default /G option.
/G1	Generates 80186/80188 instructions.
/G2	Generates 80286 instructions.
/Gc	Specifies use of PL/M- or Pascal-style function calling and naming conventions.
/Gd	Specifies standard (default) C calling conventions.
/Ge	Enables calls to the stack-checking routine (default).
/Gr	Enables the new <code>_fastcall</code> function calling conventions for eligible functions. When possible, values are passed in registers instead of on the stack.
/Gs	Suppresses generation of calls to the stack-checking routine.
/Gt <i>[number]</i>	Places data items greater than or equal to <i>number</i> bytes in a new segment. Default is 256 if no number is specified.
/Gu	When used with /ND, sets DS to DGroup for calls to <code>_cdecl</code> functions.
/Gw	Generates entry/exit code sequences suitable for use in Microsoft Windows applications.
/GW	Same as /Gw, but generates more efficient entry sequences. Used for code other than user callback function.
/GY	Generates overlay-compatible code, which conforms to requirements of the CTOS Virtual Code Segment Management facility. Combines /r, /Or, and /t. Incompatible with /Oy. Requires far code pointers (/AI).
/H <i>number</i>	Restricts external names to <i>number</i> significant characters. The default, and maximum, is 31 characters.

Command Reference

Option	Description
/HELP	Calls the QuickHelp utility. If the QuickHelp program is not available, CL displays the most commonly used options to the standard output. This option is not case sensitive.
/I <i>directory</i>	Adds <i>directory</i> to the beginning of the list of directories to be searched for include files.
/J	Changes the default for char type from signed to unsigned.
/Jm	Processes string constants as multi-byte strings. A multibyte character is one with a first byte value ranging between 0x81-0xa0 or 0xe0-0xfc. If such a leadbyte value is encountered, both the leadbyte and the character following the leadbyte are retained in the string constant. This special processing avoids the misinterpretation of the second byte of a multibyte character as a leading backslash of a control character escape sequence.
/Lp	Causes the linker to create a protected-mode executable file.
/Lr	Causes the linker to create a real-mode executable file.
/link <i>link-libinfo</i>	Passes linker options or library names in <i>link-libinfo</i> to MLINK. If specified, must be the last input on the command line.
/MAMASM_ <i>option</i>	Passes the specified options to the Microsoft Macro Assembler (MASM). MASM is automatically invoked for files listed on the command line with the extension .ASM.
/ND <i>dataseg</i>	Sets the data segment name. See also /Gu.
/NM <i>module</i>	Sets the module name.
/nologo	Suppresses the display of the sign-on banner.
/NT <i>textseg</i>	Sets the code segment name.

Option

Description

/O[opt_codes]

Controls optimization. When no codes are included, default optimization is enabled. The optional *opt_codes* argument may contain one or more of the following characters:

Code	Description
a	Assumes no aliasing.
c	Enables default (block-level) local common expressions.
d	Disables all optimizations.
e	Enables global register allocation.
g	Enables global optimizations and global common expressions.
i	Enables generation of intrinsic routines.
l	Enables loop optimizations.
n	Disables unsafe loop optimizations (default).
p	Improves consistency in floating-point calculations.
r	Disables in-line returns from functions.
s	Favors smaller code size.
t	Favors faster execution speed (default).
u	Suppresses saving and restoring SI and DI registers at function prolog and epilog if unused by the function.
w	Assumes no aliases except across function calls.
x	Maximizes optimizations (equivalent to <i>/Oecilgt/Gs</i>).
y	Removes stack frame; suppresses stack frame generation for functions without arguments and automatic storage.
z	Enables maximum loop and global-register allocation optimization.

Command Reference

Option	Description
<i>/P</i>	Preprocesses the source file and sends output to a file having the base name of the source file and the extension.i (<i>basename.i</i>)
<i>/r</i>	Disables register variables. Disables <i>/Oc</i> optimization. Incompatible with <i>/Oe</i> and <i>/Og</i> .
<i>/Sl linewidth</i>	Sets the line width of source listing in characters per line. Range is 79-132. Default is 79.
<i>/Sp pagelength</i>	Sets the page length of source listing in lines per page. Range is 15-255. Default is 63.
<i>/Ss subtitle</i>	Specifies <i>subtitle</i> for source listing.
<i>/St title</i>	Specifies <i>title</i> for source listing.
<i>/t</i>	Makes all static functions PUBLIC.
<i>/Ta asm_srcfile</i>	Specifies that <i>asm_srcfile</i> is to be treated as an assembler source file, whether or not it has an .ASM extension.
<i>/Tc c_srcfile</i>	Indicates that <i>c_srcfile</i> is a C source file, whether or not it has a .c extension.
<i>/u</i>	Removes (undefines) definitions of all predefined identifiers.
<i>/U identifier</i>	Removes the definition of the given predefined identifier.
<i>/V string</i>	Copies the version <i>string</i> to the object file.
<i>/w</i>	Suppresses compiler warning messages; same as <i>/W0</i> .
<i>/W{01121314}</i>	Sets the output level for compiler warning messages. The default is 1.
<i>/WX</i>	Makes all warnings fatal; no object file is generated when a warning occurs.
<i>/X</i>	Ignores the INCLUDE environment variable setting when searching for include files.

Option	Description
<i>/Za</i>	Enforces American National Standards Institute (ANSI) language compatibility, disabling extensions specific to Microsoft C.
<i>/Zc</i>	Causes functions declared as <code>_pascal</code> and <code>_plm</code> to be treated without regard to case.
<i>/Ze</i>	Enables CTOS Microsoft C extensions. This is the default <i>/Z</i> option.
<i>/Zg</i>	Generates function prototypes from function definitions and writes declarations to standard output, without compiling the program.
<i>/Zi</i>	Generates symbolic information required by the CodeView debugger.
<i>/ZI</i>	Suppresses emission of library search records in the object file.
<i>/Zp[<i>{1 2 4}</i>]</i>	Packs structure members on the specified byte boundary. The default is <i>/Zp2</i> .
<i>/Zs sourcefiles</i>	Performs a syntax check only.

CodeView

Summary The CTOS Microsoft CodeView debugger runs compiled programs while simultaneously displaying source code, variables, memory locations, processor registers, and other pertinent information.

Syntax The CodeView debugger has two Executive command forms.

```
CV  
    [options] runfile [arguments]
```

Note: *[arguments] specifies command-line arguments for your program. To specify these arguments in CodeView, click on Run, Set Runtime Arguments.*

```
CV RUN  
    [[options] runfile ]  
    [Case]  
    [Command]  
    [Parameter 1]  
    [Parameter...]  
    [Parameter 16]
```

Note: *CV RUN works like the CTOS Run command. Supply your application parameters in the parameter fields.*

Environment Variables Recognized

```
HELPPFILES  
INIT  
PATH
```

Command Line Options

Option	Description
/B	Starts in black-and-white mode.
/Ccommands	Executes commands on start up.
/L	Enables debugging of a dynamic-linked library.
/M	Disables CodeView support of the mouse (use this option when debugging an application that supports the mouse)
/TSF	Toggles the Statefileread file switch in TOOLS.INI to the opposite setting, which causes CV to either read or ignore CURRENT.STS on startup. For example, if Statefileread is set to Yes, /TSF toggles it to No, causing CV to ignore CURRENT.STS, and visa versa. When CV reads CURRENT.STS, it returns startup screens to how they last appeared when debugging.

CodeView Commands

These commands affect CodeView operation.

Action	Keyboard	Mouse
Display Help about the selected topic.	F1	Click Help
Display contents of Help	Shift+F1	Click Help, <Contents>
Go to next Help screen	Code+F1	-----
Go to previously viewed Help screen	Copy+F3	Click <Back> button on help screen
Toggle register window	F2	Click View, Registers
Toggle source/assembly/mixed modes	F3	Click Options, Source Windows
Toggle memory window formats	Shift+F3	Click Options, Memory Window
Switch to output screen	F4	Click View, Output
Close window	Code+F4	Click on button in upper left of window
Go to next Breakpoint or to program end	F5	Click Left on Go on status line
Switch to next window	F6	Click desired window
Switch to previous window	Shift+F6	Click desired window
Execute to cursor	F7	Click Right at location on status line
Trace into procedure	F8	Click Left on Trace

Action	Keyboard	Mouse
Change window size	Code+F8	Click Left on window border and drag
Toggle Breakpoint at line with cursor	F9 at location	Double click Left at location and drag
Step over procedure	F10	Click Left on Step
Maximize window	Code+F10	Click on button in upper right corner of window
Change flag in Register window	Any printing character	Double click left on flag
Move to next command (Command window only)	Tab	Click Left at location
Move to previous command (Command window only)	Shift+Tab	Click Left at location
Find selected text	Code+\	Click Search, Selected Text
Repeat last find	Copy+/\	Click Search, Repeat Find
Add Watch expression	Code+W	Click Watch, Add Watch
Delete Watch expression	Code+U	Click Watch, Delete Watch
Open Quick Watch window for a variable	Shift+F9	Click Watch, Quick Watch

Dialog Commands

Enter these commands in the Command window.

Name	Syntax	Description
<i>address</i>	[{ <i>segment</i> <i>/register</i> };] <i>offset (type*)</i> <i>constant</i>	Identifies the location of an expression in various commands
Add Watch	W? <i>expression</i> [, <i>format</i>]	Displays <i>expressions</i> or memory range in the Watch window
Assemble	A [<i>address</i>]	Assembles mnemonics starting at <i>address</i>
Breakpoint Clear	BC [<i>list</i> !*]	Clears breakpoints in <i>list</i> or in all breakpoints (*)
Breakpoint Disable	BD [<i>list</i> !*]	Turns off breakpoints in <i>list</i> or in all breakpoints (*)
Breakpoint Enable	BE [<i>list</i> !*]	Enables breakpoints in <i>list</i> or in all breakpoints (*)
Breakpoint List	BL	List breakpoints with the status of each
Breakpoint Set	BP [<i>address</i>] [= <i>symbol</i> [<i>range</i>]] [<i>?expression</i>] [, <i>passcount</i>] [, <i>*commands</i>]	Break execution with <i>address</i> is reached Breaks execution when the value of expressions changes; if <i>address</i> is listed, the expression is evaluated only at that address Break execution when expression is true; if <i>address</i> is listed, the expression is evaluated only at that address

Name	Syntax	Description
Comment	*comment	Displays explanatory text
Compare Memory	C <i>range address</i>	Compares <i>bytes</i> in <i>range</i> with bytes beginning at <i>address</i> ; displays any mismatched pairs
Current Location	.	Displays the current location
Delay	:	Delays execution of redirected commands (may be repeated for longer delays)
Delete Watch	Y { <i>number</i> !}	Deletes (yanks) Watch statements
Display Expression	? <i>expression</i> [, <i>format</i>]	Displays <i>expression</i> in <i>format</i>
Dump	D[<i>type</i>] [<i>address</i> / <i>range</i>]	Dumps memory address or range in <i>type</i> format
Enter	E [<i>type</i>] <i>address</i> [<i>list</i>]	Enters memory value in <i>type</i> format
Examine Symbols	X[LI*!]?[<i>module</i> .] [<i>function</i> .] [<i>symbol</i>] [*]	Displays specified symbols
Execute	E	Executes in slow motion
Fill Memory	F <i>range list</i>	Fills addresses in <i>range</i> with values in <i>list</i>
Go	G [<i>breakaddress</i>]	Executes to <i>breakaddress</i> or to end
List Watch	W	Lists current Watch statements

Command Reference

Name	Syntax	Description
Move Memory	M <i>range address</i>	Copies values in <i>range</i> memory block to <i>address</i>
Options	O [<i>option</i> [+ -]]	Views or sets CodeView Options, including bytes coded (B), flip/swap (F), case sensitivity (C), show symbol address (L), or symbols (S)
Pause	"	Interrupts execution of redirected commands and waits for keystroke
Port Input	I <i>port</i>	Reads and displays byte from <i>port</i>
Port Output	O <i>port byte</i>	Sends <i>byte</i> to <i>port</i>
Program Step	P [<i>count</i>]	Executes source lines or instructions, stepping over routine, procedure, and interrupt calls; repeats <i>count</i> times
Quick Watch	?? <i>symbol</i>	Displays local variables and complete data structures in a dialog box
Quit	Q	Exits and returns to the Executive
Radix	N [<i>radixnumber</i>]	Sets input radix
Redirection	[[[T]> >] < =] <i>devicename</i>	Redirects input or output to or from <i>devicename</i>
Redraw	@	Redraws the screen
Register	R [<i>registername</i> [[= <i>expression</i>]]]	Displays registers and flags, or sets new registers and flags

Name	Syntax	Description
Restart	L [<i>arguments</i>]	Restarts program
Screen Exchange	\ [<i>time</i>]	Exchanges the CodeView and output screens
Search	/ [<i>regularexpression</i>]	Searches for a regular expression
Search Memory	S <i>range list</i>	Searches <i>range</i> for values in <i>list</i>
Source Display Mode	S [+l-l&]	Sets display mode to source, assembly, or mixed
Stack Trace	K	Displays active routines on the stack
Tab Set	# <i>number</i>	Sets <i>number</i> spaces for each tab character
Trace	T [<i>count</i>]	Executes source lines or instructions, tracing into routine, procedure, or interrupt calls; repeats <i>count</i> times
Unassemble	U [<i>viewaddress</i>]	Displays assembly-language instructions
View	V [[<i>viewaddress</i>] . <i>linenumber</i>]	Displays source lines
8087	7	Displays 80x87 registers

Size Specifiers

Use these data types with Dump and Enter dialog commands.

Data Type	Description
None	Default type
A	ASCII (8-bit) characters
B	Byte (8-bit) hexadecimal values
D	Double-word (32-bit) hexadecimal values
I	Signed integer (16-bit) decimal values; equivalent to C signed int
L	Long (64-bit) floating point (real) values; equivalent to C double
S	Short (32-bit) floating point values; equivalent to C float
T	10-byte (80-bit) floating point values; equivalent to C long double
U	Unsigned integer (16-bit) decimal values; equivalent to C unsigned int
W	Word (16-bit) hexadecimal values

Format Specifiers

Character	Description
d	Signed decimal integer
i	Signed decimal integer
u	Unsigned decimal integer
o	Unsigned octal integer
x X	Hexadecimal integer

Character	Description
f	Signed value in floating point decimal format with six decimal places
e E	Signed value in scientific notation format with up to six decimal places (trailing zeros and decimal joint truncated)
g G	Signed value with floating point decimal or scientific notation format, whichever is more compact
c	Single character
s	Characters printed up to the first null character

Environment Service

Summary ENVIRONMENT SERVICE, an overpacked CTOS utility, manages memory areas where applications may obtain configuration data, post and retrieve inter-application messages, or use as auxiliary storage. These areas, referred to as environments contain a series of null-terminated strings called environment variables.

For more information, refer to Section 5, Building Applications. For complete information on how to use this product, refer to your *CTOS Environment Service Administration and Programming Guide*.

Command ENVIRONMENT CLEAR
 ENVIRONMENT COPY
 ENVIRONMENT QUERY
 ENVIRONMENT RELOAD
 ENVIRONMENT SET
 ENVIRONMENT STATUS
 ENVIRONMENT VERSION
 INSTALL ENVIRONMENT SERVICE
 DEINSTALL ENVIRONMENT SERVICE

Syntax ENVIRONMENT SET
 [Variable = value]
 [Global Environment]

 EnvironmentConfig.Sys

 :INCLUDE:[sys]<mscIncludes>
 :CL:/Fc /Fs /Al /c /Gs /Zp1
 :QH:[sys]<Msc>
 :TMP:[!m0]<\$>

Variable, Examples, and Commands

Variable	Example SET	Use With
CL	CL= <i>/options</i>	C
HELPPFILES	HELPPFILES=[d0]<mschelp>	All except CL, MLIB, MLINK, and PWBRMAKE
INCLUDE	INCLUDE=[d0]<mscincludes>	CL PWB
INIT	INIT=[d0]<msc>	CL PWB
LIB	LIB=[d0]<msclibs>	MLINK PWB
MAKEFLAGS	MAKEFLAGS=Y	NMAKE
MLINK	MLINK= <i>/options</i>	MLINK
PATH	PATH=[d0]<msc>; [d0]<working>	CV PWB
QH	QH=[d0]<msc>	QH
TMP	TMP=[d0]<sys>	CL HELPMMAKE MLIB PWB

Variable Descriptions

Variable	Description
CL	Compiler Options. If present, prepends to command line options specified.
HELPPFILES	Path to search for Help files (.Hlp).
INCLUDE	Search path(s) for include files (.h).
INIT	Path were the Programmer's WorkBench and the CTOS Microsoft CodeView debugger will look for TOOLS.INI and CURRENT.STS at startup. The default is the current working directory.
LIB	Search path(s) for libraries (.Lib).
MAKEFLAGS	NMAKE startup options. Specify options without slashes. For example, MAKEFILES=Y always causes NMAKE to create a submit file rather than spawn.
MLINK	MLINK options (appended to command line options). M
PATH	Search path(s) for source files and the CodeView debugger debuggee files.
QH	Search path for QuickHelp files. QH differs from HELPPFILES in that QH also searches paths specified in other environment variables.
TMP	Path to store temporary files. The default is the current working directory.

HELPMAKE

Summary The HelpMake utility creates Help files and customizes the Help files supplied with CTOS Microsoft C. It creates a Help database from one or more input files that contain information specially formatted for the Help system. If stack errors occur, use option /V0.

Syntax HELPMAKE
[options] {/E[n] | /D[n]} sourcefiles

You must supply the /HELP, /E (encode), or /D(decode) option.

Environment Variables Recognized

HELPPFILES
TMP

Command Line Options

Option	Description
/Ac	Specifies c as an application-specific control character for the Help database, marking a line that contains special information for internal use by the application.
/C	Indicates that the context strings are case sensitive. At run time, all searches for Help topics are case sensitive.

Option	Description								
<code>/D[letter]</code>	Decodes the input file into its component parts. If a destination file is not specified with the <code>/O</code> option, the Help file is decoded to stdout. HelpMake decodes the file in different ways, depending on the letter specified.								
	<table><tr><th>Letter</th><th>Effect</th></tr><tr><td><code>/D</code></td><td>Decode. Fully decodes the Help database, leaving all cross references and formatting information intact.</td></tr><tr><td><code>/DS</code></td><td>Decode Split. No decompression occurs. Not compatible with other options. Splits the concatenated, compressed Help database into its components, using their original names. If the database was created without concatenation (the default), HelpMake copies it to a file with its original name.</td></tr><tr><td><code>/DU</code></td><td>Decode Unformatted. Decompresses the database and removes all screen formatting and cross references. The output can still be used later for input and recompression, but all of the screen formatting and cross references are lost.</td></tr></table>	Letter	Effect	<code>/D</code>	Decode. Fully decodes the Help database, leaving all cross references and formatting information intact.	<code>/DS</code>	Decode Split. No decompression occurs. Not compatible with other options. Splits the concatenated, compressed Help database into its components, using their original names. If the database was created without concatenation (the default), HelpMake copies it to a file with its original name.	<code>/DU</code>	Decode Unformatted. Decompresses the database and removes all screen formatting and cross references. The output can still be used later for input and recompression, but all of the screen formatting and cross references are lost.
Letter	Effect								
<code>/D</code>	Decode. Fully decodes the Help database, leaving all cross references and formatting information intact.								
<code>/DS</code>	Decode Split. No decompression occurs. Not compatible with other options. Splits the concatenated, compressed Help database into its components, using their original names. If the database was created without concatenation (the default), HelpMake copies it to a file with its original name.								
<code>/DU</code>	Decode Unformatted. Decompresses the database and removes all screen formatting and cross references. The output can still be used later for input and recompression, but all of the screen formatting and cross references are lost.								

Option	Description												
/E[n]	Creates (encodes) a Help database from a specified text file (or files). The optional <i>n</i> indicates the amount of compression to take place. If <i>n</i> is omitted, HelpMake compresses the file as much as possible, thereby reducing the size of the file by about 50 percent. The more compression requested, the longer HelpMake takes to create a database file. The value of <i>n</i> is a number in the ranges 0-15. It is the sum of successive powers of 2 representing these compression techniques: <table> <tr> <th>Value</th><th>Technique</th></tr> <tr> <td>0</td><td>No compression</td></tr> <tr> <td>1</td><td>Run-length compression</td></tr> <tr> <td>2</td><td>Keyword compression</td></tr> <tr> <td>4</td><td>Extended keyword compression</td></tr> <tr> <td>8</td><td>Huffman compression</td></tr> </table> Add values to combine compression techniques. For example, to get run-length and keyword compression, use /E3.	Value	Technique	0	No compression	1	Run-length compression	2	Keyword compression	4	Extended keyword compression	8	Huffman compression
Value	Technique												
0	No compression												
1	Run-length compression												
2	Keyword compression												
4	Extended keyword compression												
8	Huffman compression												
/H	Displays a summary of HelpMake syntax and then exits.												
/HELP	Calls the QuickHelp utility. If the QuickHelp program is not available, HelpMake displays the most commonly used HelpMake options to the standard output (without encoding or decoding any files).												
/K <i>filename</i>	Specifies a file containing word-separator characters. This file should consist of a single line of text containing characters that separate words. ASCII characters from 0 to 32 (including the space), and character 127, are always separators. If the /K option is not specified, the following characters are also considered separators: <pre>!"#\$%&'()*+,-./:;<=>?@[\\^_`{ }~</pre>												
/L	Locks the generated file so that it cannot be decoded by HelpMake at a later time.												

Option	Description								
/Ooutfile	Specifies <i>outfile</i> as the name of the Help database. The name <i>outfile</i> is optional with the /D option.								
/Sn	Specifies the type of input file, according to the following <i>n</i> values: <table><tr><th>Option</th><th>File Type</th></tr><tr><td>/S1</td><td>Rich Text Format (RTF)</td></tr><tr><td>/S2</td><td>QuickHelp Format (default)</td></tr><tr><td>/S3</td><td>Minimally Formatted ASCII</td></tr></table>	Option	File Type	/S1	Rich Text Format (RTF)	/S2	QuickHelp Format (default)	/S3	Minimally Formatted ASCII
Option	File Type								
/S1	Rich Text Format (RTF)								
/S2	QuickHelp Format (default)								
/S3	Minimally Formatted ASCII								
/T	During encoding, translates dot commands to application-specific commands. During decoding, translates application commands to dot commands.								

Option	Description																		
<code>/V[n]</code>	Indicates the verbosity of diagnostic and informational output, depending on the value of <i>n</i>. If you omit this option or specify only <code>/V</code>, HelpMake gives you its most verbose output. If stack errors occur, specify no verbosity with <code>/V0</code>. The possible values of <i>n</i> appear in this list: <table> <tr> <th>Option</th><th>Effect</th></tr> <tr> <td><code>/V</code></td><td>Maximum diagnostic output</td></tr> <tr> <td><code>/V0</code></td><td>No diagnostic output and no banner</td></tr> <tr> <td><code>/V1</code></td><td>Prints only HelpMake banner and fatal error messages (default)</td></tr> <tr> <td><code>/V2</code></td><td>Prints pass names</td></tr> <tr> <td><code>/V3</code></td><td>Prints contexts on first pass</td></tr> <tr> <td><code>/V4</code></td><td>Prints contexts on each pass</td></tr> <tr> <td><code>/V5</code></td><td>Prints any intermediate steps within each pass</td></tr> <tr> <td><code>/V6</code></td><td>Prints statistics on Help file and compression</td></tr> </table>	Option	Effect	<code>/V</code>	Maximum diagnostic output	<code>/V0</code>	No diagnostic output and no banner	<code>/V1</code>	Prints only HelpMake banner and fatal error messages (default)	<code>/V2</code>	Prints pass names	<code>/V3</code>	Prints contexts on first pass	<code>/V4</code>	Prints contexts on each pass	<code>/V5</code>	Prints any intermediate steps within each pass	<code>/V6</code>	Prints statistics on Help file and compression
Option	Effect																		
<code>/V</code>	Maximum diagnostic output																		
<code>/V0</code>	No diagnostic output and no banner																		
<code>/V1</code>	Prints only HelpMake banner and fatal error messages (default)																		
<code>/V2</code>	Prints pass names																		
<code>/V3</code>	Prints contexts on first pass																		
<code>/V4</code>	Prints contexts on each pass																		
<code>/V5</code>	Prints any intermediate steps within each pass																		
<code>/V6</code>	Prints statistics on Help file and compression																		
<code>/Wn</code>	Indicates the fixed width of the resulting Help text in number of characters. The value of <i>n</i> can range from 11 to 255. If <code>/W</code> is omitted, the default is 76. When encoding RTF source (<code>/S1</code>), HelpMake automatically formats the text to <i>n</i>. When encoding QuickHelp (<code>/S2</code>) or minimally formatted ASCII (<code>/S3</code>) files, HelpMake truncates lines to <i>n</i> characters.																		

MLIB

Summary The MLIB utility allows you to create, organize, and maintain run-time libraries.

Syntax MLIB
 inlibrary [options] [commands]
 [, [listfile] [, [outlibrary]]] [;]

***Note:** An alternative to [options] is to use a response file, ~filename, which contains a list of command-line arguments. Specify arguments in the same order as on the command line. New lines are equal to spaces.*

Environment Variables Recognized

TMP

Command Line Options

Option	Description
/HELP	Displays online Help for MLIB. First /HELP attempts to execute the QuickHelp utility. If QuickHelp or its database is unavailable, /HELP lists the MLIB options to the standard output.
/I[GNORECASE]	Ignore case when comparing symbols (the default). Use to combine a library marked /NOI with an unmarked library for a new unmarked library.
/NOE[XTDICTIONARY]	Do not create an extended dictionary.
/NOI[GNORECASE]	Do not ignore case when comparing symbols.
/NOL[OGO]	Suppress the sign-on banner.
/PA[GESIZE]:n	Specifies the library-page size of a new library, or changes the library-page size of an existing library. The default page size for a new library is 16 bytes.

Command	Description
<code>+filename</code>	Appends an object file or library file to the given library.
<code>-filename</code>	Deletes a module from a library.
<code>++filename</code>	Replaces a module by deleting it from the library and appending to the library an object file with the same name.
<code>*filename</code>	Extracts a module without deleting it from the library and saves the module as an object file with the same name (copies it).
<code>-*filename</code>	Extracts a module and deletes it from the library after saving it in an object file with the same name (moves it).

MLINK

Summary The MLINK utility provides a command line interface to the CTOS Linker. The C compiler, CL, automatically chains to MLINK unless you compile with the `/c` option (do not link).

Syntax

```
MLINK
    [options] objfiles [, [runfile]] [, [mapfile]]
    [, [libraries]]] . [;]
```

An alternative to [options] is to use an MLINK response file (MRF), ~MRFfilename, which contains a list of command line arguments. Specify arguments in the same order as on the command line. New lines are equal to spaces. MRF format is:

```
[objectfiles] (use '+' at end to continue
               objects)
[runfile]
[mapfile]
[libraryfiles] (use '+/' at end to continue
               libraries)
```

Note: *[options] are not case sensitive, and can appear anywhere on the command line.*

Environment Variables Recognized

MLINK
LIB

Command Line Options

Option	Description
/AS	Small model libraries.
/AC	Compact model libraries.
/AL	Large model libraries.
/AM	Medium model libraries.
/AP	<i>/AP[PENDFILE];appendfile</i> Specifies [File to append] linker argument.
/CH	<i>/CH[ARSET];character_code_set</i> Specifies a Nationalization <i>character_code_set</i> . Same as CTOS Linker config option :CharacterCodeSet:.
/CL	<i>/CL[ASSORDER];list_of_class_names</i> Specifies preferred segment ordering within left and right parenthesis. Same as CTOS Linker config option :ClassOrder:.
/CO	<i>/CO[DEVIEW]</i> Creates the .Cdv file used by the CTOS Microsoft CodeView debugger. Specifies CodeView for the linker argument [Source Debugger]. If you compile with option /Zi (compile without /c), MLINK uses this option automatically.
/CR	<i>/CR[IGHT]</i> Adds the Unisys copyright statement to the Run file. Same as linker option [Copyright notice?] Yes.

Option	Description
/CS	/CS[TRING]:<i>string</i> Specifies the copyright <i>string</i> to add to the .Run file. Same as CTOS Linker config option :CopyRight:.
/CT	/CT[OSLIBS] Specifies CTOS libraries to the linker.
/DEF	/DEF[AULTCONFIGFILE] This option is equivalent to the CTOS Linker's :DefaultConfigFile: option.
/DET	/DET[AILS] Writes link details to .Map file. Same as CTOS Linker config option :Details: Yes.
/DS	/DS[ALLOCATE] Directs the linker to load all data starting at the high end of the data segment instead of at the low end. By default, DS Allocation is off, being incompatible with CTOS Microsoft C's DGroup architecture. Same as linker option [DS allocation?] Yes.
/EXT	EXT[ENSION] Creates a PWB extension: specifies the startup modules MC0CXT.Obj and EXTHDRP.Obj; a run file extension of .Cxt; a zero-sized stack; the compact C library, CLIBCP.LIB, and the extension library, EXTSUP.LIB.
/FAR	/FAR[CALLTRANSLATION] Translates far calls/jumps to near calls/jumps when appropriate. Same as CTOS Linker config option :FarCallTranslation: Yes.
/FI	/FI[LECONFIG]:<i>file_name</i> Specifies a Linker config <i>file_name</i>. Same as linker option [Config file].

Command Reference

Option	Description
/FP8	/FP8[7] Use 80x87 floating point coprocessor, and libraries 87.LIB+mLIBFP.LIB.
/FPA	/FPA[LT] Use alternate math library, mLIBFA.LIB.
/FPE	/FPE[M] Use 80x87 floating point emulation, and libraries EM.LIB+mLIBFP.LIB.
/HEA	/HEA[P]: <i>heapsize</i> or /HEA[P]:MAXVAL Specifies the heap size.
/HEL	/HEL[P] Provides Help about MLINK options and command syntax. Displays all MLINK options.
/INF	/INF[ORMATION] Displays a completed CTOS Linker form, and the contents of the Linker config file produced.
/LIBD	/LIB[IR]: <i>path</i> Specifies path for startup object modules and default libraries.
/LIBR	/LIBR[EF] Linker config :LibraryReference: name :librefname
/LINE	/LINE[NUMBERS] Includes source-file line numbers and associated addresses in the .Map file. Same as linker option [Line numbers?] Yes.
/LINK	/LINK[COMMAND]: <i>linker_command_name</i> Names the CTOS Executive command for the CTOS Linker. If this option is not used, MLINK assumes a command of MCBIND.

Option	Description
/MAXM	/MAXM[EM]:<i>data_bytes code_bytes</i> Maximum memory array above the highest memory address of a task. Specifies CTOS Linker's [Max array, data] arguments.
/MCOH	/MCOH[EAP] Use the MCOHEAP.Obj startup module. Same as /NOSTACKSHARE.
/MCON	/MCON[RT] Use the MCONRT.Obj startup module. Same as /NORUNTIME.
/MINM	/MINM[EM]:<i>data_bytes code_bytes</i> Minimum memory array above the highest memory address of a task. Specifies CTOS Linker's [Min array, data] arguments.
/MINR	/MINR[UNTIME] Use MIN.Obj flavor module. For details, refer to Section 5, Building Applications.
/MU	/MU[LTIDEFSOK] CTOS Linker config :MultipleDefSymbolsOK: Yes
/NOC	/NOC[LIBS] This option causes MLINK to eliminate all CTOS Microsoft C elements from the link.
/NOD	/NOD[EFAULTLIBRARYS] [:<i>libame</i> [:<i>libname</i> [. . .]]] Do not use default CTOS libraries, CTOS.Lib and CTOSToolkit.Lib. If <i>libname</i> is specified, it must be the name of one of the default libraries--xLIBCP.Lib, LIBCEXT.Lib, CTOS.Lib, or CTOSToolKit.Lib--to exclude from the library list passed to the CTOS Linker. The extension .Lib is optional. See also /NOFLOAT.

Option	Description
/NOFA	NOFA[RCALLTRANSLATE] This option turns off far call translation and is the opposite of the /FARCALLTRANSLATE option.
/NOFL	NOFL[OAT] Exclude the default floating-point library list passed to the CTOS Linker, EM.LIB and xLIBFP.LIB.
/NOI	/NOI[GNORECASE] Distinguish between uppercase and lowercase letters. Same as CTOS Linker config option :CaseSensitive: Yes.
/NOL	/NOL[OGO] Suppresses the MLINK sign-on banner.
/NOM	/NOM[AP] No .Map list file
/NONC	/NONC[ONTIGUOUSOK] OK to collect segments in DGroup. Same as CTOS Linker config option :NonContiguousGroupOK: Yes.
/NOR	/NOR[UNTIME] Uses the MC0NRT.Obj startup module. For details, refer to Section 5, Building Applications. Same as /MC0NRT.
/NOST	/NOST[ACKSHARE] Use the MC0HEAP.Obj startup module. Same as /MC0HEAP.
/NOSY	/NOSY[M] No .Map list file.
/NOV	/NOV[ERSION] Inhibit MLINK's generation of the default version time stamp.

Option	Description
/NOW	/NOW[ARNINGS] Suppresses all CTOS Linker warning information. Same as CTOS Linker config option :SuppressWarnings:Yes.
/OBJ	/OBJ[DIR]:<i>path</i> Specifies path of user's object file directory.
/PACKC	/PACKC[ODE]:[<i>size</i>] Groups neighboring code segments together where size specifies the maximum size of the resultant code segment. Same as CTOS Linker config option :PackCode:Yes. When used with /FARCALLTRANSLATION, produces smaller and faster code.
/PU	/PU[BLICS] Lists publics in .Map file. Same as linker option [Publics?] Yes.
/RE	/RE[ALHUGE] Uses the REALHUGE.Obj flavor module. For details, refer to Section 5, Building Applications.
/RU	/RU[NFILEMODE]:<i>modestring</i> Specifies runfile mode, where modestring is a CTOS Linker runfile mode. Protected is the default. Same as linker option [Run file mode].
/SE	/SE[GMENTS]:<i>number</i> Specifies the number of copies of MCMEMORY.Obj to link. Each copy declares 32 segments.
/SH	/SH[OWLINKFORM] Same as /INFORMATION, but suppresses linking. Displays a completed CTOS Linker command form.
/ST	/ST[ACK]:<i>stacksize</i> Sets the stack size.

Command Reference

Option	Description
/SY	<i>/SY[MBOLFILE]:filename</i> Specifies a symbol file name. The default is First_Objfile.Sym. Same as linker option [Symbol file].
/U	<i>/U[NDEFINED]:symbol_name</i> Enters the specified <i>symbol_name</i> as an unresolved external before any object modules are opened. If <i>symbol_name</i> is not declared public or communal in any object modules in the object modules link, this causes a library search to link in the appropriate object module within the library to resolve the external. Same as CTOS Linker config option :Undefined:.
/V8	<i>/V8</i> Creates V8 Run files.
/VE	<i>/VE[RSION]:version_string</i> Specifies <i>version_string</i> in the run file header. If <i>version_string</i> has embedded spaces, enclose "<i>stringa stringb</i>" in quotes. Defines public array sbVerRun initialized to <i>version_string</i>. If the first character is %, version string is passed to the C Library function strftime as a date/time format specifier.
/XCVC	<i>/XCVC[ODE]:number</i> The number of sectors allocated for :MaxCodeViewCode: :sectorcount
/XCVL	<i>/XCVL[INES]:number</i> :MaxCodeViewLines: sectorcount
/XEX	<i>/XEX[PORTDATA]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxExportData:.
/XIM	<i>/XIM[PORTDATA]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxImportData:.

Option	Description
/XLD	<i>/XLD[CT]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxRgldct:.
/XPD	<i>/XPD[HCG]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxRgPdhCG:.
/XRG	<i>/XRG[RLEPSTUB]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxRgRlePStub:.
/XRL	<i>/XRL[EPSTUB]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxRgRle:.
/XRQ	<i>/XRQ[LABLE]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxRgRqlable:.
/XST	<i>/XST[RINGSTOC]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxStringsTOC:.
/XV8	<i>/XV8[STRINGS]:number</i> number of sectors allocated for internal Linker work area. Same as CTOS Linker config option :MaxV8Strings:.

NMAKE

Summary The NMAKE utility automates the process of compiling and linking project files by reading commands and data that you enter in description files.

Syntax NMAKE
 [options] [macrodefinitions] [targets]

Environment Variables Recognized

MAKEFLAGS

Creates macros from them. For details, refer to Inherited Macros in Section 10, Managing Development Projects with NMAKE.

Command Line Options

Option	Description
/A	Executes commands to build all the targets requested even if the they are out-of-date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from appearing.
/D	Displays the modifications date of each file when the date is checked.
/E	Causes environment variables to override macro definitions within descriptions files.
/F <i>filename</i>	Specifies <i>filename</i> as the name of the description file to use. If /F is not specified, NMAKE uses MAKEFILE as the description file.
/HELP	Calls the QuickHelp utility. If QuickHelp is unavailable, NMAKE displays the most commonly used NMAKE options to the standard output.

Option	Description
/I	Ignores exit codes (also called return or error codes) returned by programs called from the NMAKE description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays the commands from the description file that NMAKE would execute, but does not execute these commands. This options is useful for checking which targets are out-of-date and for debugging description files.
/NOLOGO	Suppresses the sign-on banner when NMAKE executes.
/P	Prints all macro definitions and target descriptions.
/Q	Returns a zero exit code if the target is up-to-date and a nonzero exit code if it is not. This option is useful when invoking NMAKE from within a batch file.
/R	Ignores inference rules and macros contained in the Tools.ini file.
/S	Suppresses display of commands as they are executed.
/T	Changes the modifications dates for outdated target files to the current date. The file contents are not modified.
/Xfilename	Sends all error output to <i>filename</i> , which can be either a file or a device. If a dash is entered instead of a file name, the error output is sent to the standard output device.
/Y	Forces NMAKE to use deferred rather than inline processing.
/Z	Internal options of use by the Programmer's WorkBench.
/?	Displays a brief summary of NMAKE syntax and exits to the operating system.

Programmer's WorkBench

Summary The Programmer's WorkBench (PWB) provides an integrated environment for developing programs in C. It runs under CTOS III and below. With PWB, you can write and edit source and other text files, define development projects, and build applications from one or more files

Syntax PWB
 [options] [files]

Environment Variables Recognized

HELPPFILES
INCLUDE
INIT
LIB
PATH
TMP

Command Line Options

Option	Description
/D [<i>init</i>]	Prevents PWB from reading initialization files, where <i>init</i> is one or more of the following characters: T Ignore TOOLS.INI S Ignore CURRENT.STAT (Implies P) P Ignore current program list /D without <i>init</i> suppresses all initialization and status files from being read at start-up.
/e <i>cmdstr</i>	Provides access to PWB functions at startup. The valid functions appear in pull down menu Options, Key Assignments. If <i>cmdstr</i> contains a space, place the string in double quotes.

Option	Description
<i>/m bookmark</i>	Positions the cursor at the file location designated by <i>bookmark</i> , instead of positioning the cursor at the last known position.
<i>/r</i>	Specifies that PWB starts in read-only mode. No editing is permitted.
<i>/t [file [/t [file]...]</i>	Indicates that any files that follow are temporary. If one <i>file</i> is specified, the editor attempts to load it. If multiple <i>files</i> are specified, the first file is loaded. When the <i>Exit</i> function is invoked, the editor saves the current file and loads the next file in the list.
<i>/?</i>	Lists command-line options for starting PWB.

PWBRMake

Summary PWBRMAKE converts x.Sbr files into database files (x.Bsc) that the Browser debugger can read in PWB. You must use this command if you command-line compile with CL and switches /Fr or /FR, which creates the x.Sbr files. If you compile with Browser option set to Generate Browse Information in PWB, you do not need to use this command. PWB automatically creates the x.Bsc file.

Syntax PWBRMAKE
[options] filename(s) . . .

Note: An alternative to [options] is to use a response file, ~filename, which contains a list of command-line arguments. Specify arguments in the same order as on the command line. New lines are equal to spaces.

Environment Variables Recognized

None

Command Line Options

Option	Description
<i>/Ei filename</i>	Omits the contents of the specified include files from the database.
<i>/Ei (filenames . . .)</i>	Supply either a single filename, or multiple filenames in parentheses (<i>file_a file_b</i>).
<i>/Em</i>	Excludes symbols in the body of macros. Use this option to include only names of macros in the Source Browser database. The default includes both names and symbols in the body of macros.
<i>/Es</i>	Omits from the database every include file specified with an absolute path name. Generally, files included with angle brackets (<code>#include <one.h></code>) are omitted from the database. However, files included within quotation marks (<code>#include "one.h"</code>) are not omitted.
<i>/HELP</i>	Displays Help for PWBRMAKE.
<i>/lu</i>	Includes unreferenced symbols. By default, PWBRMAKE does not record any symbols that are defined but not referenced. Use the <i>/lu</i> option to include unreferenced symbols in the database.
<i>/n</i>	Forces a nonincremental build. Use this option to force a full build of the database even if a .Bsc file already exists, and to prevent the .Sbr files from being truncated.
<i>/o filename</i>	Directs output to <i>filename</i> . If this option is omitted, the database file name is created by changing the extension on the first .Sbr file to .Bsc.
<i>/v</i>	Provides verbose output. The output includes the name of each .Sbr file being processed and statistics about the complete PWBRMAKE run.

QuickHelp

Summary QuickHelp displays Help files, and provides a major source of information about CTOS Microsoft C. To specify options, use QH environment variables, command line switches, or both. When QuickHelp starts, it first processes environment variables, then command-line switches. The following table lists and briefly describes the options available.

Syntax QH
 [options] topic

Environment Variables Recognized

HELPPFILES
 QH

Command Line Options

Option	Description
/d <i>filename</i>	Specifies either a specific database name or a path where the databases are found. If <i>filename</i> is specified, then that database is loaded. If a path is specified, all files with the extensions .HLP are loaded. Instead of a path, you can specify an environment variable by preceding it with a dollar sign and following it with a colon. For example, /d \$INCLUDE:*.HLP.
/l <i>number</i>	Specifies the number of lines that the QuickHelp window should copy. If you specify more lines than the current screen mode allows, QuickHelp uses the maximum number allowed by the current screen mode.
/p <i>filename</i>	Sets the name of the paste file. This option must be followed by a fully qualified file name. This option acts in the same way as the Rename Paste File command on the File menu. The default paste file is PASTE.QH, and the file is placed in the TMP directory.

Command Reference

Option	Description
<i>/pa [filename]</i>	Specifies that pasting operations are to be appended to the current paste file (rather than overwriting the file). You can follow this option with the name of a file if you do not want past operations to go to the default file PASTE.QH in the TMP directory.
<i>/r command</i>	Specifies the command that QuickHelp should execute when the Right mouse button is pressed. The default action is to simulate double-clicking the Left mouse button. In other words, single-clicking the Right mouse button is identical to double-clicking the Left mouse button. The following commands are available to change this behavior:

Command	Meaning
l	Pressing the Right mouse button will display the last topic viewed. This is identical to the View Last command on the View menu.
i	Pressing the Right mouse button will display a history of the last topics viewed. This identical to the View History command on the View menu.
w	Pressing the Right mouse button will temporarily hide the QuickHelp window, allowing you to select a topic form the screen that was displayed prior to activating QuickHelp. This is identical to the Hide Window command on the View menu.
b	Pressing the Right mouse button will display the historically previous topic.
e	Pressing the Right mouse button will continue the search for a topic--displaying the next topic found, if there are any additional topics with the same name. This is identical to the Continue Search command on the View menu.
t	Pressing the Right mouse button will display the table of contents for the current topic. This is identical to the Contents command on the View menu.

Option	Description								
/s	Specifies that clicking the mouse above or below the scroll box causes QuickHelp to scroll by lines rather than by pages.								
/t <i>name</i>	Directs QuickHelp to copy the specified section of the given topic to the current paste file. This option must be followed by a section name. If the paste-file mode is Append (/pa), QuickHelp displays the specified topic in the window. If the paste-file mode is Overwrite (/p), QuickHelp exits immediately after copying the section to the paste file. The following list describes the possible topic sections: <table> <tr> <th>Topic</th><th>Description</th></tr> <tr> <td>All</td><td>Specifies the entire topic</td></tr> <tr> <td>Syntax</td><td>Specifies the syntax section of the topic</td></tr> <tr> <td>Example</td><td>Specifies the example of the topic</td></tr> </table> When this option is specified, QuickHelp does not display its window. Instead, it searches for the topic specified on the command line, pastes the topic to the paste file, and exits. .	Topic	Description	All	Specifies the entire topic	Syntax	Specifies the syntax section of the topic	Example	Specifies the example of the topic
Topic	Description								
All	Specifies the entire topic								
Syntax	Specifies the syntax section of the topic								
Example	Specifies the example of the topic								
/u	Specifies that QuickHelp is being run by a utility. If the topic specified on the command line is not found, QuickHelp immediately exits with an exit mode of 3.								

**This Page is Blank
Do Not Print**

Appendix C

Implementation-Defined Behavior

The American National Standards Institute (ANSI) Standard for the C programming language contains an appendix called Portability Issues. The ANSI appendix lists areas of the ANSI-defined C language left open for each language implementation.

This appendix describes how CTOS Microsoft C handles these implementation-defined areas with headings in the same order as the ANSI Standard appendix. Immediately following each heading, you will find the associated ANSI chapter and section in **bold italics**.

This appendix covers the following topics:

- Translation
- Environment
- Identifiers
- Characters
- Integers
- FloatingPoint Math
- Arrays and Pointers
- Registers
- Structures, Unions, Enumerations, and Bit Fields Enumerations
- Qualifiers
- Declarators
- Statements
- Preprocessing Directives
- Library Functions

Translation

Translation refers to diagnostics.

Diagnostics

How a diagnostic is identified (§2.1.1.3)

CTOS Microsoft C produces error messages in the form:

filename(line-number) : diagnostic Cnumber message

Parameter	Description
<i>filename</i>	Source file that contains the error
<i>line-number</i>	Line number that contains the error
<i>diagnostic</i>	Either error or warning
<i>number</i>	Unique four-digit number (preceded by a C) that identifies the error or warning
<i>message</i>	An explanatory message

Environment

The Environment topic is divided into these subtopics:

- Arguments to main
- Interactive devices

Arguments to main

The semantics of the arguments to main() (§2.1.2.2)

In CTOS Microsoft C, the function `main` is called at program start-up. `main` requires no prototype declaration, and can be defined with zero, two, or three parameters:

```
int main(void)
int main(int argc, char *argv[ ])
int main(int argc, char *argv[ ], char *envp[ ])
```

The example where `main` accepts three parameters is a Microsoft extension to the ANSI Standard. The third parameter, `envp`, is an array of pointers to environment variables, and is terminated by a null pointer. For more information about `main` and `envp`, refer to online Help.

The variable `argc` never holds a negative value.

The array of strings ends with `argv[argc]`, which contains a null pointer.

All elements of the `argv` array are pointers to strings.

A program invoked with no command line arguments receives a value of one for `argc`, as the name of the command used to execute the program is placed in `argv[0]`. If the command name is not available, the string `Run` is placed in `argv[0]`. Strings pointed to by `argv[1]` through `argv[argc-1]` represent program parameters.

The parameters `argc` and `argv` are modifiable and retain their last-stored values between program start-up and program termination.

Interactive Devices

What constitutes an interactive device (§2.1.2.3)

CTOS Microsoft C defines the keyboard and video as interactive devices.

Identifiers

This topic covers these subtopics:

- Significant Characters without External Linkage
- Significant Characters with External Linkage
- Uppercase and Lowercase

Significant Characters without External Linkage

The number of significant characters without external linkage (§3.1.2)

Identifiers are significant to 31 characters. The compiler does not restrict the number of characters you can use in an identifier; it simply ignores any characters beyond the limit.

Uppercase and Lowercase

Whether case distinctions are significant (§3.1.2)

CTOS Microsoft C treats identifiers within a compilation unit as case sensitive. The CTOS Linker ignores case, making externally linked identifiers case insensitive.

Thus, symbols in source files are sensitive to case. By default, symbols in object files are not.

Two CL command line options affect case sensitivity:

/Gc (Generate Pascal-style function calls) Converts all external identifiers (including function names) to uppercase.

The `_pascal` declarator performs the same operation on a function-by-function basis.

/Zc (Compile case insensitive) Converts all identifiers (excluding function names) to uppercase.

Characters

The Characters topic is divided into these subtopics:

- The ASCII Character Set
- Multibyte Characters
- Bits per Character
- Character sets
- Unrepresented Character Constants
- Wide Characters
- Converting Multibyte characters
- Range of Character Values

Multibyte Characters

Shift states for multibyte characters (§2.2.1)

Multibyte characters are used by some implementations to represent foreign-language characters not represented in the base character set.

Bits per Character

Number of bits in a character (§2.2.4.2)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The `LIMITS.H` file defines `CHAR_BIT` as 8.

Character Sets

Mapping members of the source character set (§3.1.3.4)

The source character set and execution character set include the ANSI ASCII characters listed in Table C-1. Escape sequences are also shown in Table C-1.

Table C-1. Source and Execution Character Sets

Escape Sequence	Character	ASCII Value
\a	Alert/bell	7
\b	Backspace	8
\f	Form feed	12
\n	Newline	10
\r	Carriage return	13
\t	Horizontal tab	9
\v	Vertical tab	11
\"	Double quotation	34
\'	Single quotation	39
\\	Backslash	92

Unrepresented Character Constants

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (§3.1.3.4)

CTOS Microsoft C does not support wide characters.

Wide Characters

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (§3.1.3.4)

CTOS Microsoft C does not support wide characters.

Converting Multibyte Characters

The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (§3.1.3.4)

CTOS Microsoft C does not support wide characters.

Range of char Values

Whether a plain char has the same range of values as a signed char or an unsigned char (§3.2.1.1)

All character values range from 0x00 to 0xFF, signed or unsigned. If a char is not explicitly marked as signed or unsigned, it defaults to the signed type.

CL option /J changes the default from signed to unsigned.

Integers

The Integers topic is divided into the following subtopics:

- Range of Integer Values
- Demotion of Integers
- Signed Bitwise Operations
- Remainders
- Right Shifts

Range of Integer Values

The representations and sets of values of the various types of integers (§3.1.2.5)

Short integers contain 16 bits (two bytes). Long integers contain 32 bits (four bytes). Signed integers are represented in two's-complement form. The most-significant bit holds these signs: 1 for negative, 0 for positive and zero. The values are listed below:

Type	Minimum and Maximum
unsigned short	0 to 65535
signed short	-32768 to 32767
unsigned long	0 to 4294967295
signed long	-2147483648 to 2147483647

Demotion of Integers

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (§3.2.1.2)

When a long integer is cast to a short, or a short is cast to a char, the least-significant bytes are retained.

For example, this line:

```
short x = (short)0x12345678L;
```

assigns the value 0x5678 to x, and this line:

```
char y = (char)0x1234;
```

assigns the value 0x34 to y.

When signed variables are converted to unsigned and vice versa, the bit patterns remain the same. For example, casting -2 (0xFE) to an unsigned value yields 254 (also 0xFE).

Signed Bitwise Operations

The results of bitwise operations on signed integers (§3.3)

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example, -16 & 99 can be expressed in binary as:

```

11111111 11110000
&00000000 01100011
-----
00000000 01100000

```

The result of the bitwise AND is 96.

Remainders

The sign of the remainder on integer division (§3.3.5)

The sign of the remainder is the same as the sign of the dividend. For example:

```

50 / -6 == -8
50 % -6 == 2
-50 / 6 == -8
-50 % 6 == -2

```

Right Shifts

The result of a right shift of a negative-value signed integral type (§3.3.7)

Shifting a negative value to the right yields the negative of half the absolute value, rounded down. For example, -253 (binary 11111111 00000011) shifted right one bit produces -127 (binary 11111111 10000001). A positive 253 shifts right to produce +126.

Right shifts preserve the sign bit. When a signed integer shifts right, the most-significant bit remains set. When an unsigned integer shifts right, the most-significant bit is cleared. Thus, if 0xF000 is signed, a right shift produces 0xF800. If 0xF000 is unsigned, the result is 0x7800. Shifting a positive number right sixteen times produces 0x0000. Shifting a negative number right sixteen times produces 0xFFFF.

Floating-Point Math

The Floating-Point Math topic is divided into the following subtopics:

- Values
- Casting Integers to Floating-Point Values
- Truncation of Floating-Point Values

Values

The representations and sets of values of the various types of floating-point numbers (§3.1.2.5)

The float type contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its range is +/- 3.4E38 with at least 7 digits of precision.

The double type contains 64 bits: 1 for the sign, 11 for the exponent, and 52 for the mantissa. Its range is +/- 1.7E308 with at least 15 digits of precision.

The long double type contains 80 bits: 1 for the sign, 15 for the exponent, and 64 for the mantissa. Its range is +/- 1.2E4932 with at least 17 digits of precision.

Casting Integers to Floating-Point Values

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (§3.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

For example, casting an unsigned long (with 32 bits of precision) to a float (whose mantissa has 23 bits of precision) rounds the number to the nearest multiple of 256. The long values 4294966913 - 4294967167 are all rounded to the float value 4294967040.

Truncation of Floating-Point Values

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (§3.2.1.4)

When an underflow occurs, the value of a floating-point variable is rounded down to zero. An overflow causes a run-time math error.

Arrays and Pointers

The Arrays and Pointers is divided into the following subtopics:

- Largest Array Size
- Casting Pointers
- Pointer Subtraction

Largest Array Size

The type of integer required to hold the maximum size of an array--that is, the size of `size_t` (§3.3.3.4, 4.1.1)

The `size_t` typedef is an unsigned short, with the range 0x0000 to 0xFFFF. Huge arrays can exceed this limit if they contain more than 65,535 elements. Arithmetic operations on huge arrays should therefore cast `size_t` and the results of an arithmetic operations on pointers to unsigned long.

Casting Pointers

The result of casting a pointer to an integer or vice versa (§3.3.4)

Near pointers are the same size as short integers; casting near to short (or short to near) has no immediate effect on the value.

Far pointers and huge pointers are the same size as long integers. Casting far/huge to long (or long to far/huge) has no immediate effect on the value.

When a near pointer is cast to a long, the 16-bit value is normalized, which means the segment (usually DS) and offset are combined to produce a 32-bit memory location.

When a far or huge pointer is cast to a short, the long value is truncated to a short.

The compiler normalizes based pointers when necessary, unless the based pointer is a constant zero, in which case it is assumed to be a null pointer.

Pointer Subtraction

The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t (§3.3.6, 4.1.1)

A ptrdiff_t is a signed integer in the range -32768 to 32767, with one exception. Because huge pointers can address more than 64K of memory, subtracting one huge pointer from another can yield a result that is a long integer. The result of subtracting two huge pointers should be cast to a long.

The result of subtracting two char pointers to a char array of more than 32,767 bytes should be cast to a long.

The compiler normalizes based pointers when necessary. In most cases, based pointers are treated as far pointers.

Registers

Registers refers to the availability of registers.

Availability of Registers

The extent to which objects can actually be placed in registers by use of the register storage-class specifier (§3.5.1)

Two registers, SI and DI, are available in CTOS Microsoft C. Register variables with a type that has 16 bits may be allocated in these registers.

Structures, Unions, Enumerations, and Bit Fields

This topic is divided into the following subtopics:

- Improper Access to a Union
- Sign of Bit Fields
- Storage of Bit Fields
- Alignment of Bit Fields
- Enum Type

Improper Access to a Union

A member of a union object is accessed using a member of a different type (§3.3.2.3)

If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.

For example, a union of float and int is declared. A float value is stored, but the program later accesses the value as an int. In this situation, the value depends on the internal storage of float values. Thus, the integer value is not reliable.

Sign of Bit Fields

Whether a plain int field is treated as a signed int bit field or as an unsigned int bit field (§3.5.2.1)

Bit fields can be signed or unsigned. Plain bit fields are treated as signed.

Storage of Bit Fields

The order of allocation of bit fields within an int (§3.5.2.1)

Bit fields are allocated within a 16-bit integer from least-significant to most-significant bit. For example, in the following code:

```
struct mybitfields
{
    unsigned a : 4;
    unsigned b : 5;
    unsigned c : 7;
} test;

void main(void)
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
```

the bits are arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since 80x86 processors store the low byte of integer values before the high byte, the integer 0x01F2 above is stored in physical memory as 0xF2 followed by 0x01.

Alignment of Bit Fields

Whether a bit field can straddle a storage-unit boundary (§3.5.2.1)

Bit fields default to size short, which can cross a byte boundary (see Storage of Bit Fields) but not a 16-bit boundary. If the size and location of a bit field cause it to overflow the current integer, the field is moved to the beginning of the next available integer.

If a bit field is declared as a long, it can hold up to 32 bits.

In either case, an individual field cannot cross a 16-bit or 32-bit boundary.

The Enum Type

The integer type chosen to represent the values of an enumeration type (§3.5.2.2)

A variable declared as enum is a signed short integer.

Qualifiers

Qualifiers refers to Access to Volatile Objects.

Access to Volatile Objects

What constitutes an access to an object that has volatile-qualified type (§3.5.3)

Any reference to a volatile-qualified type is an access.

Declarators

Declarators refers to the maximum number of declarators.

Maximum Number

The maximum number of declarators that can modify an arithmetic, structure, or union type (§3.5.4)

CTOS Microsoft C does not limit the number of declarators. The number is limited only by available memory.

Statements

Statements refers to the limits on switch statements.

Limits on Switch Statements

The maximum number of case values in a switch statement (§3.6.4.2)

CTOS Microsoft C does not limit the number of case values in a switch statement. The number is limited only by available memory.

Preprocessing Directives

The Preprocessing Directives topic is divided into the following subtopics:

- Character Constants and Conditional Inclusion
- Including Bracketed File Names
- Including Quoted File Names
- Character Sequences
- Pragmas
- Default Date and Time

Character Constants and Conditional Inclusion

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value (§3.8.1)

The character set used in preprocessor statements is the same as the execution character set. The preprocessor recognizes negative character values.

Including Bracketted File Names

The method for locating includable source files (§3.8.2)

The preprocessor first searches the directories specified by CL option /I. If /I is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. If more than one directory appears as part of the /I option or within the INCLUDE variable, the preprocessor searches them in the order they appear.

For example, the command:

```
CL  
  [Args]  /I [Sys] <MscIncludes> myprog.c
```

causes the preprocessor to search the directory [Sys]<MscIncludes> for include files such as STDIO.H.

The commands:

```
SET ENVIRONMENT  
  [Variable=value]  INCLUDE = [Sys] <MscIncludes>  
  
CL  
  [Args]  myprog.c
```

have a similar effect.

If both sets of searches fail, a fatal error is generated.

Including Quoted File Names

The support for quoted names for includable source files (§3.8.2)

If the file name is fully specified, with a path that includes a volume name (for example, [Sys]<Special>ORANGE.H), the preprocessor uses that path.

If the file name is not fully specified, the preprocessor searches the directory of the file that included it.

If the include file is not found there, the rules for bracketed file names apply.

Character Sequences

The mapping of source file character sequences (§\s3.8.2)

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported.

Thus, to specify an include file whose name contains a backslash, use one backslash only:

```
#include "sub \myfile"
```

Within source code, two backslashes are necessary:

```
fil = fopen("sub\\myfile", "rt");
```

Pragmas

The behavior on each recognized #pragma directive (§3.8.6)

The following pragmas are defined in online Help:

#pragma alloc_text	#pragma linesize	#pragma pagesize
#pragma check_pointer	#pragma loop_opt	#pragma same_seg
#pragma check_stack	#pragma message	#pragma skip
#pragma comment	#pragma optimize	#pragma subtitle
#pragma function	#pragma pack	#pragma title
#pragma intrinsic	#pragma page	

Default Date and Time

The definitions for `_DATE_` and `_TIME_` when, respectively, the date and time of translation are not available (§3.8.8)

When a hardware clock is not accessible, the default values for `_DATE_` and `_TIME_` are Friday, May 3, 1957 and 5:00 PM.

Library Functions

The Library Functions topic is divided into these subtopics:

- NULL Macro
- Diagnostic Printed by the assert Function
- Character Testing
- Domain Errors
- Underflow of FloatingPoint Values
- fmod Function
- Terminating Newline Characters
- Blank Lines
- Null Characters
- File Position in Append Mode
- Truncation of Text Files
- File Buffering
- ZeroLength Files
- File Names
- File Access Limits
- Deleting Open Files
- Renaming with a Name that Exists
- Printing Pointer Values

- Reading Pointer Values
- Reading Ranges
- File Position Errors
- Messages Generated by perror Function
- Allocating Zero Memory
- abort Function
- atexit Function
- Environment Names
- system Function
- strerror Function
- Time Zone
- clock Function

NULL Macro

The null pointer constant to which the macro NULL expands
(§4.1.5)

Several include files define the NULL macro as ((void *)0).

Diagnostic Printed by the assert Function

The diagnostic printed by and the termination behavior of the assert function (§4.2)

The assert function prints a diagnostic message and calls the abort routine if the expression is false (0). The diagnostic message has the form:

Assertion failed: [*expression*], file [*filename*], line [*linenumber*]

where:

filename Name of the source file

linenumber Line number of the assertion that failed in the source file

No action is taken if *expression* is true (nonzero).

Character Testing

The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions (§4.3.1)

Function	Tests For
isalnum	Characters 0-9, A-Z, a-z ASCII 48-57, 65-90, 97-122
isalpha	Characters A-Z, a-z ASCII 65-90, 97-122
iscntrl	ASCII 0-31, 127
islower	Characters a-z ASCII 97-122

Function	Tests For
isprint	Characters A-Z, a-z, 0-9, punctuation, space ASCII 32-126
isupper	Characters A-Z ASCII 65-90

Domain Errors

The values returned by the mathematics functions on domain errors (§4.5.1)

The ERRNO.H file defines the domain error constant EDOM as 33.

Underflow of Floating-Point Values

Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors (§4.5.1)

A floating-point underflow does not set the expression errno to ERANGE. When a value approaches zero and eventually underflows, the value is set to zero.

fmod Function

Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero (§4.5.6.4)

When the fmod function has a second argument of zero, the function returns zero.

Terminating Newline Characters

Whether the last line of a text stream requires a terminating newline character (§4.9.2)

Stream functions recognize either newline or end-of-file as the terminating character for a line.

Blank Lines

Whether space characters that are written out to a text stream immediately before a newline character appear when read in (§4.9.2)

Space characters are preserved.

Null Characters

The number of null characters that can be appended to data written to a binary stream (§4.9.2)

Any number of null characters can be appended to a binary stream.

File Position in Append Mode

Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (§4.9.3)

When a file is opened in append mode, the file position indicator initially points to the end of the file.

Truncation of Text Files

Whether a write on a text stream causes the associated file to be truncated beyond that point (§4.9.3)

Writing to a text stream does not truncate the file beyond that point.

File Buffering

The characteristics of file buffering (§4.9.3)

Disk files accessed through standard I/O functions are fully buffered. By default, the buffer holds 512 bytes.

Zero-Length Files

Whether a zero-length file actually exists (§4.9.3)

Files with a length of zero are permitted.

File Names

The rules for composing valid file names (§4.9.3)

A file specification can include an optional volume name (between square brackets), an optional directory name (between angle brackets), and a file name. For example: [Sys]<Sys>myfile.

Volume names and directory names can contain up to 12 characters. Case is ignored.

File names can contain up to 50 characters. You may add your own extension as part of the 50 characters, for example: myfile.test. Case is ignored. CTOS utility programs generate their own extensions and suffixes, for example: myfile.test-Old. The characters {} [] < > / \ , + - = & * ~ and ? are permitted within the name or extension. However, using these characters in a filename is not recommended since it may cause difficulties with utilities which interpret these characters in a special way. You can use @, but not as the first character.

File Access Limits

Whether the same file can be open multiple times (§4.9.3)

Opening a file that is already open is not permitted.

Deleting Open Files

The effect of the remove function on an open file (§4.9.4.1)

The remove function deletes open files only.

Renaming with a Name that Exists

The effect if a file with the new name exists prior to a call to the rename function (§4.9.4.2)

If you attempt to rename a file using a name that exists, the rename function fails and returns an error code.

Printing Pointer Values

The output for %p conversion in the fprintf function (§4.9.6.1)

CTOS Microsoft C supports three types of pointer conversions: %p (a pointer), %lp (a 32-bit far pointer), and %hp (a 16-bit near pointer).

The fprintf function produces hexadecimal values of the form XXXX (an offset) for near pointers or XXXX:XXXX (a segment plus an offset, separated by a colon) for far pointers. The output for %p depends on the memory model in use.

Reading Pointer Values

The input for %p conversion in the fscanf function (§4.9.6.2)

When the %p format character is specified, the fscanf function converts pointers from hexadecimal ASCII values into the correct address.

Reading Ranges

The interpretation of a dash (-) character that is neither the first nor the last character in the scanlist for %[conversion in the fscanf function (§4.9.6.2)

The following line:

```
fscanf(fileptr, "%[A-Z]", strptr);
```

reads any number of characters in the range A-Z into the string to which strptr points.

File Position Errors

The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure (§4.9.9.1, 4.9.9.4)

When `fgetpos` or `ftell` fails, `errno` is set to the manifest constant `EINVAL` if the position is invalid or `EBADF` if the file number is bad. The constants are defined in `ERRNO.H`.

Messages Generated by the `perror` Function

The messages generated by the `perror` function (§4.9.10.4)

The `perror` function generates these system messages for the last C library call that produced the error.

Error Number	Error Message Text
0	Error 0
2	No such file or directory
7	Arg list too long
8	Exec format error
9	Bad file number
12	Not enough core
13	Permission denied
17	File exists
18	Cross-device link
22	Invalid argument

Error Number	Error Message Text
24	Too many open files
28	No space left on device
33	Math argument
34	Result too large
36	Resource deadlock would occur

Omitted error numbers have no message text associated with them.

Allocating Zero Memory

The behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero (§4.10.3)

The `calloc`, `malloc`, and `realloc` functions accept zero as an argument. No actual memory is allocated, but the memory size can be modified later by `realloc`.

abort Function

The behavior of the `abort` function with regard to open and temporary files (§4.10.4.1)

The `abort` function does not close files that are open or temporary. It does not flush stream buffers.

atexit Function

The status returned by the `atexit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE` (§4.10.4.3r)

The `atexit` function returns zero if successful, or a nonzero value if unsuccessful.

Environment Names

The set of environment names and the method for altering the environment list used by the getenv function (§4.10.4.4)

The set of environment names is unlimited.

To change environment variables from within a C program, call the putenv function. To change environment variables from the CTOS Executive command line, use the SET ENVIRONMENT command; for example:

```
SET ENVIRONMENT  
  [Variable=value]    LIB= [Vol] <LIBS>
```

Changes made by the putenv function last only until the program ends.

system Function

The contents and mode of execution of the string by the system function (§4.10.4.5)

The system function does not work with CTOS but returns 0 and sets errno to ENOENT whether or not the argument passed is NULL.

strerror Function

The contents of the error message strings returned by the strerror function (§4.11.6.2)

The `strerror` function maps error numbers to error-message strings as follows.

Error Number	Error Message Text
0	Error 0
2	No such file or directory
7	Arg list too long
8	Exec format error
9	Bad file number
12	Not enough core
13	Permission denied
17	File exists
18	Cross-device link
22	Invalid argument
24	Too many open files
28	No space left on device
33	Math argument
34	Result too large
36	Resource deadlock would occur

Omitted error numbers have no message text associated with them.

Time Zone

The local time zone and Daylight Saving Time (§4.12.1)

The local time zone is Pacific Standard Time. CTOS Microsoft C supports Daylight Saving Time.

Clock Function

The era for the clock function (§4.12.2.1)

The clock function's era begins (with a value of 0) when the C program starts to execute. It returns times measured in 1/1000th seconds.

Appendix D

Compiler Limits and Numerical Ranges

This appendix contains reference tables that show the following information:

- Compiler limits:
 - Limits imposed by the C compiler
 - Program limits at run time
- Numerical ranges:
 - Data ranges defined in `LIMITS.H`
 - Numerical values defined in `FLOAT.H`

Compiler Limits

Table D-1 shows compiler limits on various C language constructs.

Table D-1. Limits Imposed by the C Compiler

Item	Limit
String literal	2048 bytes, including the terminating null character ('\0')
Constant	Determined by its type; refer to online Help.
Identifier	31 bytes (additional characters are discarded)
Declarations	15 levels of nesting for structure and union definitions
Macro definition	6K bytes, 255 formal arguments
Macro expansion	6K
Preprocessor arguments	3K (approximately)
if, #ifdef, and #ifndef directives	16 levels of nesting
Include files	12 levels of nesting
Initialization	30 levels of nesting

Table D-2 shows various run-time limits.

Table D-2. Program Limits at Run Time

Item	Limit
File size	2 ³² - 1 bytes (4 gigabytes)
Open files (streams)	20*
Command line	64K
Command line and environment table	64K, combined

* The compiler opens stdin, stdout, and stderr when your program begins, leaving 17 streams.

Numerical Ranges

Table D-3 shows data ranges for constants defined in `LIMITS.H`.

Table D-3. Data Ranges Defined in `LIMITS.H`

Manifest Constant	Description	Value
<code>CHAR_MAX</code>	Maximum char value	127
<code>CHAR_MIN</code>	Minimum char value	-127
<code>CHAR_MAX</code>	Maximum char value	255
<code>CHAR_MIN</code>	Minimum char value	0
<code>SCHAR_MAX</code>	Maximum signed char value	127
<code>SCHAR_MIN</code>	Minimum signed char value	-127
<code>UCHAR_MAX</code>	Maximum unsigned char value	255
<code>CHAR_BIT</code>	Number of bits in a char	8
<code>USHRT_MAX</code>	Maximum unsigned short value	65535
<code>SHRT_MAX</code>	Maximum (signed) short value	32767
<code>SHRT_MIN</code>	Minimum (signed) short value	-32768
<code>UINT_MAX</code>	Maximum unsigned int value	4294967295
<code>ULONG_MAX</code>	Maximum unsigned long value	4294967295

continued

Table D-3. Data Ranges Defined in LIMITS.H (cont.)

Manifest Constant	Description	Value
INT_MAX	Maximum (signed) int value	32767
INT_MIN	Minimum (signed) int value	-32767
LONG_MAX	Maximum (signed) long value	2147483647
LONG_MIN	Minimum (signed) long value	-2147483647

Note: The two sets of values for `CHAR_MAX` and `CHAR_MIN` are defined within an `#ifndef` block as follows:

```

ifndef_CHAR_UNSIGNED
#define CHAR_MAX    127
#define CHAR_MIN    (-127)
#else
#define CHAR_MAX    255
#define CHAR_MIN    0
#endif

```

Compiler Limits and Numerical Ranges

Table D-4 shows numerical values for constants defined in `FLOAT.H`.

Table D-4. Numerical Values Defined in `FLOAT.H`

Manifest Constant	Description	Value
<code>DBL_DIG</code>	Number of decimal digits of precision	15
<code>DBL_EPSILON</code>	Smallest value such that $1.0 + \text{DBL_EPSILON} \neq 1.0$	2.2204460492503131e-016
<code>DBL_MANT_DIG</code>	Number of bits in mantissa	53
<code>DBL_MAX</code>	Maximum value	1.7976931348623158e+308
<code>DBL_MAX_10_EXP</code>	Maximum decimal exponent	308
<code>DBL_MAX_EXP</code>	Maximum binary exponent	1024
<code>DBL_MIN</code>	Minimum positive value	2.2250738585072014e-308
<code>DBL_MIN_10_EXP</code>	Minimum decimal exponent	307
<code>DBL_MIN_EXP</code>	Minimum binary exponent	-1021
<code>FLT_DIG</code>	Number of decimal digits of precision	7

continued

Table D-4. Numerical Values Defined in FLOAT.H (cont.)

Manifest Constant	Description	Value
FLT_EPSILON	Smallest value such that $1.0 + \text{FLT_EPSILON} \neq 1.0$	1.192092896e-07F
FLT_MANT_DIG	Number of bits in mantissa	24
FLT_MAX	Maximum value	3.402823466e+38F
FLT_MAX_10_EXP	Maximum decimal exponent	38
FLT_MAX_EXP	Maximum binary exponent	128
FLT_MIN	Minimum positive value	1.175494351e-38F
FLT_MIN_10_EXP	Minimum decimal exponent	-37
FLT_MIN_EXP	Minimum binary exponent	-125
LDBL_DIG	Number of decimal digits of precision	19
LDBL_EPSILON	Smallest value such that $1.0 + \text{LDBL_EPSILON} \neq 1.0$	5.4210108624275221706 e-020
LDBL_MANT_DIG	Number of bits in mantissa	64

continued

Table D-4. Numerical Values Defined in FLOAT.H (cont.)

Manifest Constant	Description	Value
LDBL_MAX	Maximum value	1.189731495357231765 e+4932L
LDBL_MAX_10_EXP	Maximum decimal exponent	4932
LDBL_MAX_EXP	Maximum binary exponent	16384
LDBL_MIN	Minimum positive value	3.3621031431120935063 e-4932L
LDBL_MIN_10_EXP	Minimum decimal exponent	-4932
LDBL_MIN_EXP	Minimum binary exponent	-16381

Appendix E

printf/scanf Format Specifiers

The format syntax for printf and scanf is, respectively:

`% [flags][width][.precision][{F | N | h | l | L}]type`

and:

`% [*][width][{F | N }][{h | l}]type`

Table E-1 describes the printf and scanf syntax fields and their respective permitted values. The pf column shows whether the field applies to printf; the sf column shows whether the field applies to scanf.

Table E-1. printf and scanf Format Specifiers

Field	Description	pf	sf
<i>flags</i>	Characters that justify output and control the printing of signs, blanks, decimal points, and octal and hexadecimal prefixes	Yes	No
	Code Description		
	- Left justifies	Yes	No
	+ Prefixes signed output with + or -; always printed with sign	Yes	No
	0 Adds leading zeros to reach minimum width	Yes	No

continued

printf/scanf Format Specifiers

Table E-1. printf and scanf Format Specifiers (cont.)

Field	Description	pf	sf
	<i>blank</i> (' ')Prefixes zero or signed positive value with a blank	Yes	No
	<i>#</i> With: o, x, X: Prefixes nonzero output value with 0, 0x, or 0X e, E, f: Inserts decimal point g, G: Inserts decimal point and does not truncate trailing zeros	Yes	No
<i>*</i>	Suppresses assignment of the next field	No	Yes
<i>width</i>	Specifies minimum width in characters For printf, if width is an asterisk(*) then the next argument--an integer--determines the width. This width argument precedes the argument being formatted.	Yes	Yes
<i>precision</i>	Specifies precision in number of digits and decimal places If precision is an asterisk(*) then the next argument--an integer--determines the precision. This precision argument precedes the argument being formatted.	Yes	No

continued

Table E-1. printf and scanf Format Specifiers (cont.)

Field	Description	pf	sf
	With:		
	d, i, u, o ,x, X: Specifies minimum number of digits		
	If number is less than precision, output value is padded on the left with zeros. Does not truncate values larger than precision.		
	e, E: Specifies number of digits after the decimal point and rounds the last printed digit		
	f: Specifies number of digits after the decimal point		
	g, G: Specifies the maximum number of significant digits		
	c: Has no effect		
	s: Specifies maximum number of characters to be printed		
F	Explicitly indicates far value	Yes	Yes
N	Explicitly indicates near value	Yes	Yes

continued

printf/scanf Format Specifiers

Table E-1. printf and scanf Format Specifiers (cont.)

Field	Description	pf	sf
h	With: d, i, o, x, X: Specifies short int u: Specifies short unsigned int	Yes	Yes
l	With: d, i, o, x, X: Specifies long int u: Specifies long unsigned int e, E, f, g, G: Specifies double	Yes	Yes
L	With e, E, f, g, G: Specifies long double	Yes	Yes
<i>type</i>	Characters that justify output and control the printing of signs, blanks, decimal points, octal and hexadecimal prefixes.	Yes	Yes

continued

Table E-1. printf and scanf Format Specifiers (cont.)

Field	Description	pf	sf
	Code Meaning		
	c Single character	Yes	Yes
	d Signed decimal integer	Yes	Yes
	e, E Exponential; case sets case of exponent key	Yes	Yes
	f Floating-point value	Yes	Yes
	g, G e or f format; case sets case of exponent key	Yes	Yes
	i For printf, signed decimal integer For scanf, signed decimal, octal, or hexadecimal integer	Yes	Yes
	n Number of bytes successfully written (printf) or read (scanf) placed in its corresponding argument, which is a pointer to an integer Performs no input or output.	Yes	Yes

continued

Table E-1. printf and scanf Format Specifiers (cont.)

Field	Description	pf	sf
o	Unsigned octal integer	Yes	Yes
p	Pointer to void; prints address pointed to by the argument	Yes	Yes
s	Null-terminated string	Yes	Yes
u	Unsigned decimal integer	Yes	Yes
x, X	For printf, unsigned hexadecimal integer using abcdef or ABCDEF For scanf, unsigned hexadecimal integer	Yes	Yes

Appendix F

Re-entrant Run-Time Library Functions

The following C run-time library functions are re-entrant:

abs	labs	memset	strcmpi	strnset
atoi	lfind	mkdir	strcpy	strchr
atol	lsearch	movedata	stricmp	strrev
bsearch	memccpy	putch	strlen	strset
chdir	memchr	rmdir	strlwr	strstr
getpid	memcmp	segread	strncat	strupr
hallo	memcpy	strcat	strncmp	swap
hfree	memicmp	strchr	strncpy	tolower
itoa	memmove	strcmp	strnicmp	toupper

This Page is Blank
Do Not Print

Appendix G

Sample NMAKE Description Files

To illustrate the power of NMAKE, this appendix contains six makefiles, from small to complex. If you use this appendix along with Section 10, Managing Development Projects with NMAKE, you will learn how to exploit the power of NMAKE faster than if you use Section 10 only.

This appendix contains examples used to build these projects:

- Help File
- Library Version String
- Programmer's WorkBench Extension
- Run File
- Library
- Run Files, System Service, and Libraries

The first example illustrates how to create a Help file. The second example illustrates how to create a library version-string macro that you may want to use for your development projects. The last example is very complex, uses the library version-string macro, and shows how to build two run files, two libraries, and a system service--all with one makefile. All examples were used various times in the CTOS Microsoft C production environment. We have included them to give you ideas on how you might use NMAKE for your own projects.

Help File

This example creates UTILS.hlp, one of the Help files included in <MSCHelp>. UTILS.hlp is composed of 4 separate Help files, appended together.

This example illustrates how to use ALL:. Without this line, NMAKE executes the first target/dependent block only, which is lib.hlp: lib.src. This example also illustrates HelpMake switches used for creating .hlp files from .src files.

To create the files listed in the makefile, execute these two commands:

```
HelpMake  
    [Args]    /ds    UTILS.hlp
```

which splits UTILS.hlp into lib.hlp, nmake.hlp, misc.hlp, and Helpmake.hlp.

```
HelpMake  
    [Args]    /d /t /olib.src lib.hlp
```

for each component file, which creates .src files that you can edit. The makefile then converts lib.src, helmake.src, nmake.src, and misc.src to .hlp files; then appends the four .hlp files to create UTILS.hlp.

```
#####
#
# Template for .src to .hlp makefiles
# This will Append separate .hlp databases to produce a single
#   FILE.Hlp
#
#
#####

.SUFFIXES: .src .hlp
LIST = lib.hlp Helpmake.hlp nmake.hlp misc.hlp
BUILDDIR=

ALL:  UTILS.hlp

lib.hlp: lib.src
    Helpmake\n/e8 /t /w77 /olib.hlp lib.src\g

Helpmake.hlp: Helpmake.src
    Helpmake\n/e8 /t /w77 /oHelpmake.hlp Helpmake.src\g

nmake.hlp: nmake.src
    Helpmake\n/e8 /t /w77 /onmake.hlp nmake.src\g

misc.hlp: misc.src
    Helpmake\n/e8 /t /w77 /omisc.hlp misc.src\g

UTILS.hlp: $(LIST)
    Delete\nUTILS.hlp\g
    Append\n@<<append.fl$
$**
<<NOKEEP
\n$@\g
```


Library Version String

This example creates a version string and appends it to your libraries. If anyone modifies your libraries, this string disappears. Any modifications of the library by either MLIB or Librarian results in this text being removed. This makes it very simple to spot bogus libraries.

The linker appends text to a run file identifying the libraries accessed to build the run file. The linker gets this text from the tail end of the library. The typical format is:

```
LIBRARY: ctos.lib  
VERSION: 12.1.0 (thursday june 27, 1991, 14:14)  
LIBRARY: ctostoolkit.lib  
VERSION: 12.1.0 (thursday june 27, 1991, 14:15)
```

To have the appropriate text appended to your libraries, insert the attached makefile command line in your makefiles after the command line that modifies the library. The last example in this appendix incorporates this makefile.

Note: *The spaces preceeding the command can be adjusted; the embedded spaces must remain.*

```
#-----
#
# Tool macros
#
APPEND      = append\n

#-----
#
#Library version string macros--note all spaces after '=' are
#significant
#
LIBVER_LIB  =LIBRARY: $(@F)
LIBVER_VER  =VERSION: $(VERSION) (%d!!*w! !*n! !*d!, !yyyy!,
!*t!:!0m!|)

#-----
#
# The Command line
#
$(APPEND) [Kbd]\n
$(LIB_OUTPUT)\n\g\n$(LIBVER_LIB)\n$(LIBVER_VER)\n\f
```

Programmer's WorkBench Extension

This example shows how the PWB Help file extension was created.

```
#####  
#  
# To create debugging versions:  
#  
# set DEBUG=<anything> in your environment, or invoke NMAKE  
# with DEBUG= on the command line.  
#  
#  
#####  
#  
# Switch and Macro definitions.  
#  
# System options.  
#  
# DEBUG macros.  
# These are switched on or off by defining or not defining  
# DEBUG=on the nmake command line or system environment.  
#  
# $(CDEBUG) = debug dependent arguments for c compile command  
#             lines  
# $(CDEBUG2) = Additional debug dependent arguments for c  
#               compile command lines this is seperated out  
#               specifically for NDEBUG, because some compiles  
#               require it regardless of debug.  
# $(LDEBUG) = debug dependent arguments for link command lines  
# $(MDEBUG) = debug dependent arguments for masm command lines  
#
```

```

!IFDEF DEBUG
    CDEBUG = /Zdi /DDEBUG=$(DEBUG) /Olwser
    LDEBUG = /CO
    MDEBUG = /ZD /ZI /DDEBUG=$(DEBUG)
    CODE = /Fc
!ELSE
    CDEBUG = /Olwser /Gs
    CDEBUG2 = -DNDEBUG
    LDEBUG =
    MDEBUG =
    CODE =
!ENDIF
#
# Tool Macro Definitions
#
ASM = masm\n
CC = CL\n

!IFDEF version
    version = 6.1.0%D|/!0d!!nnn!!yy!_!0t!:!0m!/|
!ENDIF

!IFDEF BUILD
    HINC = $(PWBINC)
    LK = Bind
    BLD =
    BLD2 = $(LIBDIR)
    ENGINE = $(LIBDIR)
    LIBDIR = $(LIBDIR)
    LIBDOS = $(LIBDOS)
    LOCALH = <PWBHELP>
    INCLUDES = $(INCLUDE)
    PWBTOOLS = $(PWBTOOLS)
    CTOSLIB = [MSC]<12.0LIBS>

```

Sample NMAKE Description Files

```
!ELSE
```

```
    LK      = MCBind
    BLD      = <PWBHLPBIN>
    BLD2     = <PWBSUPBIN>
    HINC     = <PWBINC>
    ENGINE   = [D0]<HELPEBLD>
    LIBDIR   = [D0]<LIB>
    LIBDOS   = [D1]<PWBLIB>
    LOCALH   = [D1]<PWBHELP>
    INCLUDES = [D0]<INCLUDES >
    PWBTOLS  = [D1]<PWBTOLS>
    CTOSLIB  = [SYS]<SYS>
!ENDIF
```

```
MAPFILE      = $(@:cxt=map)
PUBLICS      =
LINENUMBERS  =
STACKSIZE    = 0
MAXARRAY     = 0 0
MINARRAY     = 0 0
RUNFILEMODE  = protected
VER          = $(version)
##
LIBRARIES = \
$(ENGINE)WHELP.LIB \
$(BLD2)extsup.lib \
$(LIBDIR)CLIBCP.LIB \
$(CTOSLIB)CTOS.LIB \
$(CTOSLIB)CTOSTOOLKIT.LIB none
```

```

DSALLOCATION = no
SYMBOLFILE   = $(@:cxt=sym)
LCONFIG      =
CODEVIEW     =

#
# Combination macros. Used to make dependancies more readable.
#
CFLAGS = -nologo $(CDEBUG) /Asun /W2 /c /Zep /D_CTOS
MFLAGS = $(MDEBUG) /ML /V /P /D_CTOS

#####
#
# target dependancies and build commands.
#
# All: build everything this makefile knows
#
all: setenv $(BLD)pwbHelp.cxt

setenv: setinc setcl setmasm
setinc:
    Environment Set\n@<<
    INCLUDE=$(LOCALH);$(HINC);$(INCLUDES);$(PWBTOOLS)
    <<NOKEEP
    \g
setcl:
    Environment Set\n@<<
    CL=$(CFLAGS)
    <<NOKEEP
    \g
setmasm:
    Environment Set\nMASM=\g

```

Sample NMAKE Description Files

```
#
# Macros to define the source and object dependencies only
# once.
# Note that because of macro replacement, spacing is VERY
# relavent
#
SRC = mhcore. mhdisp. mhevt. mhutil. mhfile. mhlook. mhwin.

##OPW = $(BLD2)EXTHDRP.Obj $(BLD)mhasm.Obj
##$(BLD)$(SRC:. =.Obj <PWBHLPBIN>)obj $(BLD)mhcw.Obj
$(BLD)mhdlg.Obj

!IFDEF BUILD
OPW = $(LIBDOS)MCOEXT.Obj \
      $(BLD2)EXTHDRP.Obj $(BLD)mhasm.Obj \
      $(BLD)$(SRC:. =.Obj )obj $(BLD)mhcw.Obj $(BLD)mhdlg.Obj

!ELSE

OPW = $(LIBDOS)MCOEXT.Obj \
      $(BLD2)EXTHDRP.Obj $(BLD)mhasm.Obj \
      $(BLD)$(SRC:. =.Obj <PWBHLPBIN>)obj $(BLD)mhcw.Obj
$(BLD)mhdlg.Obj
!ENDIF

#####
$(BLD)pwbHelp.cxt: $(OPW) $(BLD2)extsup.lib $(ENGINE)WHELP.LIB
Del\n$@\g
$(LK)\n @<<$(*).Objfls
$(OPW)
<<NOKEEP
\n$@\n$(MAPFILE)\n$(PUBLICS)\n\
$(LINENUMBERS)\n$(STACKSIZE)\n$(MAXARRAY)\n$(MINARRAY)\n\
$(RUNFILEMODE)\n'$(VER)'\n@<<$(*).libfls
$(LIBRARIES)
<<NOKEEP
!IFDEF BUILD
\n$(DSALLOCATION)\n$(SYMBOLFILE)\n$(LEGALESE)\n\
!ELSE
\n$(DSALLOCATION)\n$(SYMBOLFILE)\n\n$(LEGALESE)\n\
!ENDIF
$(LCONFIG)\n$(CODEVIEW)\g
```

```
#####
#
# Individual source dependancies.
#
INCEXTEN= $(HINC)mh.h $(HINC)Help.h $(HINC)ext.h

#
$(BLD)mhcore.Obj: mhcore.c $(INCEXTEN)
$(BLD)mhcv.Obj : mhcv.c $(INCEXTEN)
$(BLD)mhdisp.Obj: mhdisp.c $(INCEXTEN)
$(BLD)mhevt.Obj : mhevt.c $(INCEXTEN)
$(BLD)mhfile.Obj: mhFILE.C $(INCEXTEN)
$(BLD)mhlook.Obj: mhlook.c $(INCEXTEN)
$(BLD)mhutil.Obj: mhutil.c $(INCEXTEN)
$(BLD)mhwin.Obj : mhwin.c $(INCEXTEN)

$(BLD)mhdlg.Obj: mhdlg.c $(INCEXTEN) Help.sdm Help.hs
$(HINC)extint.h
    Set Environment\nCL=\g
    $(CC) -DNDEBUG -DOS2 -G2 $(CDEBUG) >$*.lst /Asnu
\n/D_CTOS/W2 /c /Zep /Fo$@ mhdlg.c\g
    Set Environment\nCL=$(CFLAGS)\g

.c{$(BLD)}.Obj:
    Delete\n$*.Obj\g
    $(CC) -DNDEBUG -DOS2 -G2 >$*.lst \n$(CODE) /Fo$@ $<\g
    Type\n$*.lst\g

$(BLD)mhasm.Obj: mhasm.Asm
    $(ASM) $(MFLAGS) /DOS2 mhasm, $(BLD)mhasm.Obj;\g
```


Run File

This example shows how Qh.Run was created, the run file for QuickHelp.

```
#####  
#  
#   CTOS version of Make file for creating CTOS QuickHelp.  
#  
#       - final version - makes qh.run only  
#       - qh is built small model  
#  
#   NOTE: problem found in optimization of qhcow.c - /Oc taken  
#   off to fix problem. 12/11/90 C6.0a but C1 not up to date.  
#   problem manifested by incorrect boxdraw code (no vertical  
#   line code)  
#  
#####  
  
CC = CL  
AS = MASM  
  
!IFDEF version  
    version = 6.1.0%D|/!0d!!nnn!!yy!_!0t!:!0m!/|  
!ENDIF  
  
# Note all spaces after '=' are significant!  
VERSION_LINE=VERSION: $(version) (%d!*w! !*n! !*d!, !yyy!,  
!*t!:!0m!|)
```

```
!IFDEF BUILD
    LK      = Bind
    INCLUDE = $(INCLUDE)
    LIBDIR  = $(LIBDIR)
    LIBDOS  = $(LIBDOS)
    LIBNOSRC = $(LIBNOSRC)
    HELPENGDIR = $(LIBDIR)
    BUILDDIR =
    CTOSLIB  = [MSC]<12.0LIBS>

!ELSE

    LK      = MCBind
    INCLUDE = [D0]<INCLUDES>
    LIBDIR  = <Lib>
    LIBDOS  = [D1]<PWBLIB>
    LIBNOSRC = [D1]<PWBLIB>
    HELPENGDIR = <Helpenbld>
    BUILDDIR = <QHBLD>
    CTOSLIB  = [SYS]<SYS>
!ENDIF
```

Sample NMAKE Description Files

```
BINDFLS =\
$(LIBDOS)mc0.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(LIBDOS)mcmemory.Obj \
$(BUILDDIR)qhasm.Obj \
$(BUILDDIR)qhfcasm.Obj \
$(BUILDDIR)qhcmp.Obj \
$(BUILDDIR)qhcow.Obj \
$(BUILDDIR)qhdisp.Obj \
$(BUILDDIR)qhdos.Obj \
$(BUILDDIR)qhfind.Obj \
$(BUILDDIR)qhHelp.Obj \
$(BUILDDIR)qhinit.Obj \
$(BUILDDIR)qhinput.Obj \
$(BUILDDIR)qhmain.Obj \
$(BUILDDIR)qhmenu.Obj \
$(BUILDDIR)qhmou.Obj \
$(BUILDDIR)qhopt.Obj \
$(BUILDDIR)qhpaste.Obj \
$(BUILDDIR)qhpseudo.Obj \
$(BUILDDIR)qhrun.Obj \
$(BUILDDIR)qhstring.Obj \
$(BUILDDIR)qhutil.Obj \
$(BUILDDIR)qhxref.Obj \
$(BUILDDIR)Helpdll.Obj \
$(BUILDDIR)natfunctions.Obj \
$(LIBDOS)apimemqh.Obj

H= qh.h qhstring.h qhcom.h qhnatdefstring.h
```

```
##### Final Build #####
```

```
# original - DOSCFLAGS = -c -W3 -Gs2 -Zlp -Ocaztge -DDOS -
nologo
# original - DOSSPACECFLAGS = -c -W3 -Gs2 -Zlp -Ocasge -DDOS -
nologo
```

```
#optimization taken out of xFLAGS
```

```
CFLAGS = -c -D_CTOS -W3 -Gs2 -Zl -Zp1 -DDOS -nologo
```

```
# Linker stuff
```

```
MAPFILE = $(:Run=map)
```

```
PUBLICS =
```

```
LINENUMBERS =
```

```
STACKSIZE = 8192 maxval
```

```
MAXARRAY = 0 0
```

```
MINARRAY = 0 0
```

```
RUNFILEMODE = protected
```

```
VER = $(version)
```

```
LIBRARIES =\
```

```
$(HELPENGDIR)SHELP.LIB\
```

```
$(LIBDIR)SLIBCP.LIB \
```

```
$(LIBDOS)DOSCALLS.LIB \
```

```
$(LIBDOS)DOSCALLS2.LIB \
```

```
$(LIBNOSRC)2.3MOUSE.LIB \
```

```
$(CTOSLIB)CTOS.LIB \
```

```
$(CTOSLIB)CTOSTOOLKIT.LIB none
```

```
DSALLOCATION = no
```

```
SYMBOLFILE = $(:Run=sym)
```

```
Default: environ $(BUILDDIR)QH.Run $(BUILDDIR)QHNatMsg.Bin
```

```
environ: setcl setinc
```

```
Set Environment\nMASM=\g
```

```
setcl:
```

```
Set Environment\nCL=$(CFLAGS)\g
```

Sample NMAKE Description Files

```
setinc:
  Set Environment\nINCLUDE=$(INCLUDE)\g
  Set Environment\n\g

$(BUILDDIR)QH.Run: $(BINDFLS)
  Del\n$@\g
  $(LK)\n@<<$(*).Objfls
$(BINDFLS)
<<NOKEEP
  \n$@\n$(MAPFILE)\n$(PUBLICS)\n\
  $(LINENUMBERS)\n$(STACKSIZE)\n$(MAXARRAY)\n$(MINARRAY)\n\
  $(RUNFILEMODE)\n'$$(VER)'\n@<<$(*).libfls
$(LIBRARIES)
<<NOKEEP
  \n$(DSALLOCATION)\n$(SYMBOLFILE)\n$(LEGALESE)\g

$(BUILDDIR)QHNatMsg.Bin: qhnatmsg.txt
  Create Message File\nqhnatmsg.txt\n$@\g
  Append\n$(LEGALESE_TXT) [Kbd]\n$@\g\n$(VERSION_LINE)\n\f

#
# These modules are optimized for speed

$(BUILDDIR)qhcmp.Obj: qhcmp.c $(H)
  Delete\n$*.Obj\g
  Delete\n$*.lst\g
  $(CC)\n >$*.lst -AS -Ocw tge \n-Fo$(BUILDDIR)qhcmp.Obj
  qhcmp.c\g
  Type\n$*.lst\g

$(BUILDDIR)qhdisp.Obj: qhdisp.c qhstring.c $(H) qhmouse.h
  Delete\n$*.Obj\g
  Delete\n$*.lst\g
  $(CC)\n >$*.lst -AS -Ocw tge \n-
  Fo$(BUILDDIR)qhdisp.Obj qhdisp.c\g
  Type\n$*.lst\g
```

```
$(BUILDDIR)qhfind.Obj: qhfind.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocwtge \n-Fo$(BUILDDIR)qhfind.Obj
    qhfind.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhHelp.Obj: qhHelp.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocwtge \n-Fo$(BUILDDIR)qhHelp.Obj
    qhHelp.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhinit.Obj: qhinit.c $(H) qh_ctos.h
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocwtge \n-Fo$(BUILDDIR)qhinit.Obj
    qhinit.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhinput.Obj: qhinput.c $(H) qhmouse.h
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocwtge \n-
    Fo$(BUILDDIR)qhinput.Obj qhinput.c\g
    Type\n$*.lst\g

$(BUILDDIR)qh mou.Obj: qhmou.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocwtge \n-Fo$(BUILDDIR)qh mou.Obj
    qhmou.c\g
    Type\n$*.lst\g
```

Sample NMAKE Description Files

```
$(BUILDDIR)qhutil.Obj: qhutil.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocw tge \n-Fo$(BUILDDIR)qhutil.Obj
    qhutil.c\g
    Type\n$*.lst\g

$(BUILDDIR)natfunctions.Obj: natfunctions.c natdefs.h
    delete\n$@\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst -AS -Ocw tge \n-
    Fo$(BUILDDIR)natfunctions.Obj
    natfunctions.c\g
    Type\n$*.lst\g

#
# These modules are optimized for space instead of speed
#

# /Oc taken off qhcow to fix optimization bug - 12/11/90
$(BUILDDIR)qhcw.Obj: qhcw.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)qhcw.Obj -AS -Owsge
    qhcw.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhdos.Obj: qhdos.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)qhdos.Obj -AS -Ocwsge
    qhdos.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhmain.Obj: qhmain.c $(H) qh_ctos.h
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)qhmain.Obj -AS -Ocwsge
    qhmain.c\g
    Type\n$*.lst\g
```

```
$(BUILDDIR)qhpseudo.Obj: qhpseudo.c $(H)
    Delete\n$.Obj\g
    Delete\n$.lst\g
    $(CC)\n >$.lst \n-Fo$(BUILDDIR)qhpseudo.Obj -AS -
    Ocwsge qhpseudo.c\g
    Type\n$.lst\g

$(BUILDDIR)qhmenu.Obj: qhmenu.c $(H)
    Delete\n$.Obj\g
    Delete\n$.lst\g
    $(CC)\n >$.lst \n-Fo$(BUILDDIR)qhmenu.Obj -AS -Ocwsge
    qhmenu.c\g
    Type\n$.lst\g

$(BUILDDIR)qhopt.Obj: qhopt.c $(H) qh_ctos.h
    Delete\n$.Obj\g
    Delete\n$.lst\g
    $(CC)\n >$.lst -AS -Ocwsge \n-Fo$(BUILDDIR)qhopt.Obj
    qhopt.c\g
    Type\n$.lst\g

$(BUILDDIR)qhpaste.Obj: qhpaste.c $(H)
    Delete\n$.Obj\g
    Delete\n$.lst\g
    $(CC)\n >$.lst \n-Fo$(BUILDDIR)qhpaste.Obj -AS -Ocwsge
    qhpaste.c\g
    Type\n$.lst\g

$(BUILDDIR)qhrun.Obj: qhrun.c $(H)
    Delete\n$.Obj\g
    Delete\n$.lst\g
    $(CC)\n >$.lst \n-Fo$(BUILDDIR)qhrun.Obj -AS -Ocwsge
    qhrun.c\g
    Type\n$.lst\g
```


Sample NMAKE Description Files

```
$(BUILDDIR)qhstring.Obj: qhstring.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)qhstring.Obj -AS - .
    Ocwsge qhstring.c\g
    Type\n$*.lst\g

$(BUILDDIR)qhhref.Obj: qhhref.c $(H)
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)qhhref.Obj -AS -Ocwsge
    qhhref.c\g
    Type\n$*.lst\g

#Helpdll brought in for 'mixed'(os/2-dos) model
$(BUILDDIR)Helpdll.Obj: Helpdll.c
    Delete\n$*.Obj\g
    Delete\n$*.lst\g
    $(CC)\n >$*.lst \n-Fo$(BUILDDIR)Helpdll.Obj -AS -Owers
    Helpdll.c\g
    Type\n$*.lst\g

#
# These are the assembly language routines
#

$(BUILDDIR)qhasm.Obj: qhasm.Asm
    Delete\n$*.Obj\g
    $(AS)\n/MU /ZI qhasm, $@;\g

$(BUILDDIR)qhfcasm.Obj: qhfcasm.Asm
    Delete\n$*.Obj\g
    $(AS)\n/ML /ZI qhfcasm, $@;\g
```

Library

This example shows how the library engine for online Help was created.

```
#####  
#  
# CTOS makefile for HELP Engine (xHELP.LIB)...this builds only  
#     small model (SHELP.LIB), for use in build of  
#     Helpmake.run and QuickHelp.run  
# Note: add msHelp:(not really a dll); really wHelp.lib, used  
#     to build pwbHelp.cxt extension  
#  
# To create debugging versions:  
#  
# set DEBUG=<anything> in your environment, or invoke NMAKE  
#     with DEBUG= on the command line.  
#####
```

Sample NMAKE Description Files

```
.ASM = masm
CC = cl
```

```
INCLUDE=<Includes>
HELPINC =<HELPINC>
```

```
#
# Switch and Macro definitions.
#
# Build Directory
#
!IFDEF BUILD
BUILDDIR=
!ELSE
BUILDDIR=<HELPEBLD>
!ENDIF
#
# System options.
#
# DEBUG macros. These are switched on or off by defining or not
#           defining DEBUG= on the nmake command line or
#           system environment.
#
# $(CDEBUG) = debug dependent arguments for c compile command
#           lines
# $(CDEBUG2) = Additional debug dependent arguments for c
#           compile command lines this is separated out
#           specifically for NDEBUG, because some compiles
#           require it regardless of debug.
# $(LDEBUG) = debug dependent arguments for link command lines
# $(MDEBUG) = debug dependent arguments for masm command lines
#
# rwh - original, to which we may return...OPTS = /Oaers
OPTS =/Owers
```

```

#!IFDEF DEBUG
#CDEBUG = /Zdi /DDEBUG=$(DEBUG)
#LDEBUG = /CO
#MDEBUG = /ZD /ZI /DDEBUG=$(DEBUG)
#DDEBUG = DEBUG=

#!ELSE

CDEBUG = $(OPTS) /Gs
CDEBUG2 = /DNDEBUG
LDEBUG =
MDEBUG =
DDEBUG =
#!ENDIF

NOLOGO =

#
#
# Combination macros. Used to make dependancies more readable.
#
# $(CFLAGS) = C compile command line flags
# $(MFLAGS) = Masm compile command line flags
#
# original CFLAGS = $(CDEBUG) $(CLIST) /W2 /c /Zep /Fo$@
# original MFLAGS = $(MDEBUG) $(MLIST) /ML /V /P
# original INCLUDE =
$(BUILDDIR);inc;$(TOOLS)\include;$(INCLUDE)
#
CFLAGS=$(NOLOGO) $(CDEBUG) $(CDEBUG2) /D_CTOS /W2 /c /Zep
MFLAGS=$(MDEBUG) /ML /V /P

H = $(HELPINC)Help.h $(HELPINC)HelpFILE.H $(HELPINC)Helpsys.h

```

Sample NMAKE Description Files

```
#####
#
default: environ $(BUILDDIR)sHelp.lib

msHelp:  environ $(BUILDDIR)wHelpp.lib

environ: setinc setcl setmasm

setinc:
  Set Environment\nINCLUDE=$(HELPINC);$(INCLUDE)\g

setcl:
  Set Environment\nCL=$(CFLAGS)\g

setmasm:
  Set Environment\nMASM=$(MFLAGS)\g

#####
#
# target dependancies and build commands.
#

#####
#
# Libraries for the various memory models : rwh - we build
# small model, for use with Helpmake and quickHelp, and a
# special wHelpp.lib for use with our pwb extensions
#
#           Used By
#           -----
# ***sHelp.lib - small model           HelpMake Utility
# mHelp.lib    - medium model
# mHelph.lib   - medium model handle/offset      Q/B
# mHelphq.lib  - medium model handle/offset /GW   Q/C
# lHelp.lib    - large model
# ***wHelpp.lib - ss!=ds model (protect mode)    msHelp.dll
#               (QH, M)
# wHelp.r.lib  - ss!=ds model (real mode)        M
# wHelpq      - ss!=ds model CODE & DATA      Q/P
#
```

```
$(BUILDDIR)SHELP.LIB: \
    $(BUILDDIR)sHelp.Obj \
    $(BUILDDIR)shinfo.Obj \
    $(BUILDDIR)shdata.Obj \
    $(BUILDDIR)shback.Obj \
    $(BUILDDIR)sHelpif.Obj \
    $(BUILDDIR)sHelpc.Obj \
    $(BUILDDIR)sHelpcel.Obj \
    $(BUILDDIR)sHelpdec.Obj \
    $(BUILDDIR)shloc.Obj \
    $(BUILDDIR)shctl.Obj \
    $(BUILDDIR)shline.Obj
Del\n$@\g
MLib\n@<<$(*).lrf
$@ $** ;
<<NOKEEP
\g
##### end small model build #####

#####
#
#
## searchpa.Obj added to supply DosSearchPath()

$(BUILDDIR)wHelpp.lib: \
    $(BUILDDIR)wHelp.Obj \
    $(BUILDDIR)whinfo.Obj \
    $(BUILDDIR)whdata.Obj \
    $(BUILDDIR)whback.Obj \
    $(BUILDDIR)wHelpif.Obj \
    $(BUILDDIR)wHelpc.Obj \
    $(BUILDDIR)wHelpcel.Obj \
    $(BUILDDIR)wHelpdll.Obj \
    $(BUILDDIR)hloc.Obj \
    $(BUILDDIR)Helpdec.Obj \
    $(BUILDDIR)lhctlp.Obj \
    $(BUILDDIR)lhlinep.Obj \
    $(BUILDDIR)searchpa.Obj
Del\n$@\g
MLib\n@<<$(*).lrf
$@ $** ;
<<NOKEEP
\g
```

Sample NMAKE Description Files

```
# keep this for documentation
# cd $(BUILDDIR)
# -$(RM) wHelpp.lib
# $(LIBR) wHelpp
#
#####

#####
#
# Individual source dependancies.
# incengine is not used as dependency
#INCENGINE= Help.c $(INCLUDE)Helpsys.h $(INCLUDE)HelpFILE.H \
#           $(INCLUDE)Help.h

#
# Small Model
#
$(BUILDDIR)shdata.Obj:      hdata.c $(H)
                        Del\n$@\g
                        $(CC)\n/AS >$*.lst /Fo$@ hdata.c\g
                        Type\n$*.lst\g
$(BUILDDIR)sHelp.Obj:      Help.c $(H)
                        Del\n$@\g
                        $(CC)\n/AS >$*.lst /Fo$@ Help.c\g
                        Type\n$*.lst\g
$(BUILDDIR)shinfo.Obj:     hinfo.c $(H)
                        Del\n$@\g
                        $(CC)\n/AS >$*.lst /Fo$@ hinfo.c\g
                        Type\n$*.lst\g
$(BUILDDIR)shback.Obj:     hback.c $(H)
                        Del\n$@\g
                        $(CC)\n/AS >$*.lst /Fo$@ hback.c\g
                        Type\n$*.lst\g
$(BUILDDIR)sHelpif.Obj:    Helpif.c $(H)
                        Del\n$@\g
                        $(CC)\n/AS >$*.lst /Fo$@ Helpif.c\g
                        Type\n$*.lst\g
```

```
$(BUILDDIR)sHelpc.Obj:    Helpcnt.c $(H)
    Del\n$@\g
    $(CC)\n/AS >$*.lst /Fo$@ Helpcnt.c\g
    Type\n$*.lst\g
$(BUILDDIR)sHelpcel.Obj:  Helpcell.c $(H)
    Del\n$@\g
    $(CC)\n/AS >$*.lst /Fo$@ Helpcell.c\g
    Type\n$*.lst\g

$(BUILDDIR)sHelpdec.Obj:  Helpdec.Asm Helpfile.inc
    Del\n$@\g
    $(AS)\n /DSMALL >$*.lst\n Helpdec.Asm,

$(BUILDDIR)sHelpdec.Obj;\g
    Type\n$*.lst\g

$(BUILDDIR)shloc.Obj:    hloc.Asm Help.inc Helpfile.inc
    Helpsys.inc
    Del\n$@\g
    $(AS)\n /DSMALL >$*.lst\n hloc.Asm,

$(BUILDDIR)shloc.Obj;\g
    Type\n$*.lst\g

$(BUILDDIR)shctl.Obj:    hctl.Asm Help.inc Helpfile.inc \
    Helpsys.inc Helpsysa.inc
    Del\n$@\g
    $(AS)\n /DSMALL >$*.lst\n hctl.Asm,

$(BUILDDIR)shctl.Obj;\g
    Type\n$*.lst\g

$(BUILDDIR)shline.Obj:   hline.Asm Help.inc Helpfile.inc \
    Helpsys.inc Helpsysa.inc
    Del\n$@\g
    $(AS)\n /DSMALL >$*.lst\n hline.Asm,

$(BUILDDIR)shline.Obj;\g
    Type\n$*.lst\g
```


Sample NMAKE Description Files

```
#####  
# Wierd model (SS!=DS) (prot)  
#  
$(BUILDDIR)searchpa.Obj:      searchpa.c  
    Del\n$@\g  
    $(CC)\n /G2 /Fo$@\n >$*.lst /NTHelp_TEXT /Asuf  
    searchpa.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)whdata.Obj:      hdata.c  $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf hdata.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)wHelp.Obj:      Help.c  $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf Help.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)whinfo.Obj:      hinfo.c  $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf hinfo.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)whback.Obj:      hback.c  $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf hback.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)wHelpif.Obj:      Helpif.c $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf Helpif.c\g  
    Type\n$*.lst\g  
  
$(BUILDDIR)wHelpc.Obj:      Helpcnt.c $(H)  
    Del\n$@\g  
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst  
    /NTHelp_TEXT /Asuf Helpcnt.c\g  
    Type\n$*.lst\g
```

```
$(BUILDDIR)wHelpcel.Obj:   Helpcell.c $(H)
    Del\n$@\g
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst
    /NTHelp_TEXT /Asuf Helpcell.c\g
    Type\n$*.lst\g

$(BUILDDIR)wHelpdll.Obj:   Helpdll.c
    Del\n$@\g
    $(CC)\n /DDSLOAD /DOS2 /G2 /Fo$@\n >$*.lst
    /NTHelp_TEXT /Asuf Helpdll.c\g
    Type\n$*.lst\g

$(BUILDDIR)lhctlp.Obj:     hctl.Asm Helpfile.inc Help.inc
    Helpsys.inc
    Del\n$@\g
    $(ASM)\n $(MFLAGS) >$*.lst\n /DOS2 hctl.Asm,

$(BUILDDIR)lhctlp.Obj;\g
    Type\n$*.lst\g

$(BUILDDIR)lhlinep.Obj:   hline.Asm Helpfile.inc Help.inc
    Helpsys.inc
    Del\n$@\g
    $(ASM)\n $(MFLAGS) >$*.lst\n /DOS2 hline.Asm,

$(BUILDDIR)lhlinep.Obj;\g
    Type\n$*.lst\g

#
$(BUILDDIR)Helpdec.Obj:   Helpdec.Asm Helpfile.inc
    Del\n$@\g
    $(ASM)\n $(MFLAGS) >$*.lst\n Helpdec.Asm,

$(BUILDDIR)Helpdec.Obj;\g
    Type\n$*.lst\g

$(BUILDDIR)hloc.Obj:   hloc.Asm Helpfile.inc Help.inc Helpsys.inc
    Del\n$@\g
    $(ASM)\n $(MFLAGS) >$*.lst\n hloc.Asm,

$(BUILDDIR)hloc.Obj;\g
    Type\n$*.lst\g
```

Run Files, System Service, and Libraries

This example shows how Task Management Service was created.

```
#
# Targets:
#
# all          Build everything this makefile knows
#              [1] tms.run
#              [2] tmsNull.run
#              [3] tmsRequest.sys
#              [4] tmsAPI.lib
#              [5] evsAPI.lib
#
# tms.run      Build the TMS system service proper
#
# tmsNull.run  Build the exit run file for "spawned" tasks
#
# tmsRequest.sys Build the loadable request file for TMS
#
# tmsAPI.lib   Build the API object library for TMS
#
# evsAPI.lib   Build the
# APIobjectlibraryforEVS(Environment Service)
#
```

```
#-----  
#  
#   To create special versions:  
#  
#   Set XXXX=<anything> in your environment, or invoke NMAKE  
#   with XXXX=on  
#   the command line. Where XXXX is:  
#  
#   SYNTAX           Checks syntax only -- no code generation.  
#  
#   LIST             Produces code listing files.  
#  
#   DEBUG            Diagnostics during development.  
#  
#   TCL              Use ".latest" compiler.  
#  
#   BROWSER          Produce source browser files -- (.sbr's &  
#   .bsc's).  
#  
#   BUILD            Used in the merged build environment.  
#  
#-----  
#
```

Sample NMAKE Description Files

```
# Path macros
```

```
#
```

```
!IFDEF BUILD
```

```
BLD_VOL          = [msc]
SRC_VOL          = [msc]
BLD_DIR          = <tms>
SRC_DIR          = <tms>
BLD_PATH         = $(BLD_VOL)$(BLD_DIR)
SRC_PATH         = $(SRC_VOL)$(SRC_DIR)
STD_INC_PATH     = $(SRC_VOL)$(INCLUDE)
STD_LIB_PATH     = $(SRC_VOL)$(LIBDIR)
SYS_LIB_PATH     = $(SRC_VOL)<12.1libs>
```

```
!ELSE
```

```
BLD_VOL          = [sansei]
SRC_VOL          = [sansei]
BLD_DIR          = <tms.o>
SRC_DIR          = <tms>
BLD_PATH         = $(BLD_VOL)$(BLD_DIR)
SRC_PATH         = $(SRC_VOL)$(SRC_DIR)
STD_INC_PATH     = $(SRC_VOL)<includes>
STD_LIB_PATH     = $(SRC_VOL)<lib>
SYS_LIB_PATH     = $(SRC_VOL)<sys>
```

```
!ENDIF
```

```
# -----
```

```
#
```

```
# Object macros
#
TMS_ENTRY_OBJ    =\
                  $(BLD_PATH)mc0nrt.Obj

TMSNULL_ENTRY_OBJ    =\
                     $(BLD_PATH)mc0nrt.Obj

TMS_OBJ          =\
                  $(BLD_PATH)tmsBsInfo.Obj\
                  $(BLD_PATH)tmsError.Obj\
                  $(BLD_PATH)tmsFileSpec.Obj\
                  $(BLD_PATH)tmsInvoke.Obj\
                  $(BLD_PATH)tmsMain.Obj\
                  $(BLD_PATH)tmsMisc.Obj\
                  $(BLD_PATH)tmsQuery.Obj\
                  $(BLD_PATH)tmsRqHandlr.Obj\
                  $(BLD_PATH)tmsScEntry.Obj\
                  $(BLD_PATH)tmsScInfo.Obj\
                  $(BLD_PATH)tmsScInst.Obj\
                  $(BLD_PATH)tmsSystem.Obj\
                  $(BLD_PATH)tmsString.Obj\
                  $(BLD_PATH)tmsTaskInfo.Obj

TMSNULL_OBJ      =\
                  $(BLD_PATH)tmsNullMain.Obj

TMS_API_OBJ      =\
                  $(BLD_PATH)tmsPipeAPI.Obj\
                  $(BLD_PATH)tmsPipeAPI2.Obj\
                  $(BLD_PATH)tmsRqLabl.Obj

EVS_API_OBJ      =\
                  $(BLD_PATH)evsRqLabl.Obj\
                  $(BLD_PATH)evsRqLablAlt.Obj
```

Sample NMAKE Description Files

```
#-----
#
# Browser macros
#
TMS_BSC      =\
              $(BLD_PATH)tmsBsInfo.sbr\
              $(BLD_PATH)tmsFileSpec.sbr\
              $(BLD_PATH)tmsInvoke.sbr\
              $(BLD_PATH)tmsMain.sbr\
              $(BLD_PATH)tmsMisc.sbr\
              $(BLD_PATH)tmsQuery.sbr\
              $(BLD_PATH)tmsRqHandlr.sbr\
              $(BLD_PATH)tmsScInfo.sbr\
              $(BLD_PATH)tmsScInst.sbr\
              $(BLD_PATH)tmsSystem.sbr\
              $(BLD_PATH)tmsTaskInfo.sbr

TMSNULL_BSC =\
              $(BLD_PATH)tmsNullMain.sbr

#-----
#
# Tool macros
#
!IFDEF BUILD
CC      = tcl
!ELSE
!   IFDEF TCL
CC      = tcl
!   ELSE
CC      = cl
!   ENDIF
LK      = bind                # Build overrides command.
!ENDIF
AS      = masm
CAS     = assemble           # CTOS Assembler.
RM      = delete
CAT     = type
VID     = video
MRS     = make request set
!IFDEF TCL
GREP    = grep
```

```

!ENDIF
LIB          = mlib
APPEND       = append
BSC          = pwbrmake
ENVSET       = environment set
ENVQRY       = environment query

```

```
MEM_MODEL    = L
```

```

#-----
#
# Filename macros
#
C_INPUT      = $(SRC_PATH)$(@B).c
ASM_INPUT    = $(SRC_PATH)$(@B).Asm
TXT_INPUT    = $(SRC_PATH)$(@B).txt

OBJ_OUTPUT   = $(@R).Obj
ERR_OUTPUT   = $(SRC_PATH)$(@B).err
LST_OUTPUT   = $(SRC_PATH)$(@B).lst
SBR_OUTPUT   = $(@R).sbr
TCL_OUTPUT   = $(SRC_PATH)$(@B).tcl
RUN_OUTPUT   = $(@R).run
SYM_OUTPUT   = $(@R).Sym
MAP_OUTPUT   = $(@R).Map
CDV_OUTPUT   = $(@R).Cdv
SYS_OUTPUT   = $(@R).sys
LIB_OUTPUT   = $(@R).lib
BSC_OUTPUT   = $(@R).bsc

#-----
#
# Command macros
#
#
#

```


Sample NMAKE Description Files

```
!IFDEF TCL
LIST=1          # Force listing output
GREG_INPUT      = $(LST_OUTPUT)
GREG_PATTERN    = si di cld
GREG_OUTPUT     = $(TCL_OUTPUT)
GREG_NAMEONLY   =
GREG_RELATIVE   =
GREG_COUNTONLY  =
GREG_CASE       = yes
GREG_EXCEPTIONS =
GREG_BLOCKNUM   =
GREG_SUPPRESS   = yes
!ENDIF

GREG_CODE       =\
!IFDEF TCL
!   IFNDEF SYNTAX
$(GREG)\n $(GREG_INPUT)\n $(GREG_PATTERN)\n
$(GREG_OUTPUT)\n\
$(GREG_NAMEONLY)\n $(GREG_RELATIVE)\n
$(GREG_COUNTONLY)\n\
$(GREG_CASE)\n $(GREG_EXCEPTIONS)\n
$(GREG_BLOCKNUM)\n\
$(GREG_SUPPRESS)\g
!   ENDIF
!ENDIF

TRASH_OLD       =\
$(RM)\n $(OBJ_OUTPUT)\g\
$(RM)\n $(ERR_OUTPUT)\g\
$(RM)\n $(LST_OUTPUT)\g

TRASH_OLD_C     =\
$(TRASH_OLD)\
!IFDEF TCL
$(RM)\n $(GREG_OUTPUT)\g\
!ENDIF
$(RM)\n $(SBR_OUTPUT)\g

TRASH_OLD_SYS   =\
$(RM)\n $(SYS_OUTPUT)\g\
$(RM)\n $(ERR_OUTPUT)\g
```

```

TRASH_OLD_RUN      =\
                    $(RM)\n $(RUN_OUTPUT)\g\
                    $(RM)\n $(MAP_OUTPUT)\g\
                    $(RM)\n $(SYM_OUTPUT)\g\
                    $(RM)\n $(CDV_OUTPUT)\g

TRASH_OLD_LIB      =\
                    $(RM)\n $(LIB_OUTPUT)\g

TRASH_OLD_BSC      =\
                    $(RM)\n $(BSC_OUTPUT)\g

CTOS_ASM           =\
                    $(TRASH_OLD)\
                    $(CAS)\n $(ASM_INPUT)\n yes\n\n $(OBJ_OUTPUT)\
                    $(LST_OUTPUT)\n\n$(ERR_OUTPUT)\n\n\n\g\
                    $(CAT)\n $(ERR_OUTPUT)\g

#-----
#
# Library version string macros -- note all spaces after '='
# are significant!
#
LIBVER_LIB=LIBRARY: $(@F)
LIBVER_VER=VERSION: $(VERSION) (%d|!*w| !*n| !*d|, !yyyy!,
!*t!:!0m!|)

#-----
#
# Video macro
#
PAUSE              = yes

```

Sample NMAKE Description Files

```
#-----
#
# Bind macros
#
PUBLICS          =
LINENUMBERS      =
TMS_STACK        = 2048
TMS_MAXARRAY     = 0 0
TMS_MINARRAY     = 0 0
TMS_RUNMODE      = gdtprotected
TMSNULL_STACK    = 1024
TMSNULL_MAXARRAY = 0 0
TMSNULL_MINARRAY = 0 0
TMSNULL_RUNMODE  = protected
!IFDEF BUILD
VERSION          = 1.1 (MSC $(version))
!ELSE
!   IFNDEF VERSION
!       IFDEF TCL
VERSION          = 1.1/TCL/%u_%d!!0o!!/!0d!_!0t!::!0m!!
!           ELSE
VERSION          = 1.1/%u_%d!!0o!!/!0d!_!0t!::!0m!!
!       ENDIF
!   ENDIF
!ENDIF
TMS_LIBRARIES    =\
$(BLD_PATH)evsAPI.lib\
$(BLD_PATH)tmsAPI.lib\
$(STD_LIB_PATH)$(MEM_MODEL)libCp.lib\
$(SYS_LIB_PATH)ctos.lib\
$(SYS_LIB_PATH)ctosToolkit.lib
TMSNULL_LIBRARIES =\
$(STD_LIB_PATH)$(MEM_MODEL)libCp.lib\
$(SYS_LIB_PATH)ctos.lib\
$(SYS_LIB_PATH)ctosToolkit.lib
DSALLOC          =
COPYRIGHT        = yes
!IFDEF BUILD
FILETOAPPEND     = $(LEGALESE_RUN)
```

```
!ELSE
FILETOAPPEND      = $(SRC_PATH)tmsLinkerConfig.sys
!ENDIF
CONFIG FILE       = $(SRC_PATH)tmsLinkerConfig.sys
!IFDEF DEBUG
SOURCEDEBUGGER    = codeView
!ELSE
SOURCEDEBUGGER    =
!ENDIF

#-----
#
# Debug macros
#
# $(CDEBUG)        = debug dependent arguments for c compile
# command lines
# $(ADEBUG)        = debug dependent arguments for masn command
# lines
#
!IFDEF DEBUG
!  IFDEF SYNTAX
CDEBUG            = /W4 /Zs /DDEBUG=1
ADEBUG            = /ZD /ZI /DDEBUG=1
!  ELSE
!    IFDEF BROWSER
CDEBUG            = /W4 /Zs /DDEBUG=1 /FR$(SBR_OUTPUT)
ADEBUG            = /ZD /ZI /DDEBUG=1
!    ELSE
CDEBUG            = /W4 /Od /G2s /Zi /DDEBUG=1
ADEBUG            = /ZD /ZI /DDEBUG=1
!    ENDIF
!  ENDIF
!ELSE
CDEBUG            = /W3 /Ocgilt /Ow /G2s /DNDEBUG=1
ADEBUG            = /DNDEBUG=1
!ENDIF

#-----
#
# Listing macros. Define LIST= to generate listings
#
```

Sample NMAKE Description Files

```
!IFDEF LIST
!   IFNDEF SYNTAX
CLIST      = /Fc$(LST_OUTPUT)
ALIST      = /LA /L
!   ENDIF
!ENDIF

#-----
#
# Combination macros. Used to make dependancies more readable.
#
# $(CFLAGS) = C compile command line flags
# $(AFLAGS) = Masm compile command line flags
#
INCLUDE_PATH      = /I$(SRC_PATH) /I$(STD_INC_PATH)
CFLAGS            = $(INCLUDE_PATH) $(CDEBUG) $(CLIST) /c
/A$(MEM_MODEL) /Zep1
AFLAGS            = $(INCLUDE_PATH) $(ADEBUG) $(ALIST) /ML /T /P
BFLAGS            = /n /o$(BSC_OUTPUT)

!IFDEF TCL
!   IFNDEF SYNTAX
CFLAGS        = $(CFLAGS) /Fo$(OBJ_OUTPUT)
!   ENDIF
!ENDIF

!IFNDEF TCL
CFLAGS        = $(CFLAGS) /nologo
!ENDIF
```

```
#-----  
#  
# Inference rules  
#  
.SUFFIXES:  
.SUFFIXES:      .c .Asm .Obj  
  
{$(SRC_PATH)}.c{$(BLD_PATH)}.Obj:  
    $(TRASH_OLD_C)  
    $(CC)\n @<<  
$(CFLAGS)  
<<  
    \n $(C_INPUT) >$(ERR_OUTPUT)\g  
    $(CAT)\n $(ERR_OUTPUT)\g  
    $(GREP_CODE)  
  
.c.Obj:  
    $(TRASH_OLD_C)  
    $(CC)\n @<<  
$(CFLAGS)  
<<  
    \n $(C_INPUT) >$(ERR_OUTPUT)\g  
    $(CAT)\n $(ERR_OUTPUT)\g  
    $(GREP_CODE)  
  
{$(SRC_PATH)}.Asm{$(BLD_PATH)}.Obj:  
    $(TRASH_OLD)  
    $(AS)\n @<<  
$(AFLAGS)  
<<  
    \n @<<  
$(ASM_INPUT),$(OBJ_OUTPUT),$(LST_OUTPUT); >$(ERR_OUTPUT)  
<<  
    \g  
    $(CAT)\n $(ERR_OUTPUT)\g
```

Sample NMAKE Description Files

```
.Asm.Obj:
    $(TRASH_OLD)
    $(AS)\n @<<
$(AFLAGS)
<<
    \n @<<
$(ASM_INPUT),$(OBJ_OUTPUT),$(LST_OUTPUT); >$(ERR_OUTPUT)
<<
    \g
    $(CAT)\n $(ERR_OUTPUT)\g
```

```
#-----
#
# target dependancies and build commands.
#
# All: build everything this makefile knows
#
all:                environment\
                    tmsAPI.lib\
                    evsAPI.lib\
                    tmsRequest.sys\
                    tms.run\
                    tmsNull.run

tms.run::           $(BLD_PATH)tms.run

tmsNull.run::       $(BLD_PATH)tmsNull.run

tmsRequest.sys:     $(BLD_PATH)tmsRequest.sys

tmsAPI.lib::        $(BLD_PATH)tmsAPI.lib

evsAPI.lib::        $(BLD_PATH)evsAPI.lib

#-----
#
```

```
#    environment
#
environment:
    $(ENVSET)\nTMP=[Scr]<$$>\g
    $(ENVSET)\nCL=\g
    $(ENVSET)\nINCLUDE=\g
    $(ENVSET)\nMASM=\g
    $(ENVQRY)\n\n\n$(BLD_PATH)BUILD_ENVIRONMENT\g

#-----
#
#    mc0nrt.Obj
#
$(BLD_PATH)mc0nrt.Obj:\
    $(SRC_PATH)mc0nrt.Asm

#-----
#
#    tms.run
#
$(BLD_PATH)tms.run::\
    $(TMS_ENTRY_OBJ)\
    $(TMS_OBJ)\
    $(TMS_LIBRARIES)\
    $(TMS_API_OBJ)
!IFDEF SYNTAX
!    IFDEF DEBUG
!        IFDEF BROWSER
            $(TRASH_OLD_BSC)
            $(BSC)\n $(BFLAGS)\n @<<
$(TMS_BSC)
<<
            \g
!        ENDIF
!    ENDIF
            $(TRASH_OLD_RUN)
            $(LK)\n @<<
$(TMS_ENTRY_OBJ)
$(TMS_OBJ)
<<
```


Sample NMAKE Description Files

```
\n $(RUN_OUTPUT)\n $(MAP_OUTPUT)\n $(PUBLICS)\n $(LINENUMBERS)\n\n $(TMS_STACK)\n $(TMS_MAXARRAY)\n $(TMS_MINARRAY)\n\n $(TMS_RUNMODE)\n\n '$(VERSION)'\n @<<
$(TMS_LIBRARIES)
none
<<
\n $(DSALLOC)\n $(SYM_OUTPUT)\n $(COPYRIGHT)\n
$(FILETOAPPEND)\n\n
$(CONFIG_FILE)\n $(SOURCEDEBUGGER)\ng
!ENDIF

TMS_COMMON_H      =\
    $(SRC_PATH)tms.h\
    $(SRC_PATH)tmsAPI.h\
    $(SRC_PATH)tmsBsInfo.h\
    $(SRC_PATH)tmsError.h\
    $(SRC_PATH)tmsFileSpec.h\
    $(SRC_PATH)tmsGlobals.h\
    $(SRC_PATH)tmsInvoke.h\
    $(SRC_PATH)tmsMisc.h\
    $(SRC_PATH)tmsQuery.h\
    $(SRC_PATH)tmsRqHandler.h\
    $(SRC_PATH)tmsScInfo.h\
    $(SRC_PATH)tmsScInst.h\
    $(SRC_PATH)tmsSystem.h\
    $(SRC_PATH)tmsTypes.h\
    $(SRC_PATH)tmsTaskInfo.h

$(BLD_PATH)tmsBsInfo.Obj:\
    $(SRC_PATH)tmsBsInfo.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h\
    $(SRC_PATH)syscom.h
```

```
$(BLD_PATH)tmsError.Obj:\
    $(SRC_PATH)tmsError.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h

$(BLD_PATH)tmsFileSpec.Obj:\
    $(SRC_PATH)tmsFileSpec.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h

$(BLD_PATH)tmsInvoke.Obj:\
    $(SRC_PATH)tmsInvoke.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h\
    $(SRC_PATH)evsAPI.h

$(BLD_PATH)tmsMain.Obj:\
    $(SRC_PATH)tmsMain.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h\
    $(SRC_PATH)syscom.h

$(BLD_PATH)tmsMisc.Obj:\
    $(SRC_PATH)tmsMisc.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h

$(BLD_PATH)tmsQuery.Obj:\
    $(SRC_PATH)tmsQuery.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosTypes.h
```

Sample NMAKE Description Files

```
$(BLD_PATH)tmsRqHandler.Obj:\
    $(SRC_PATH)tmsRqHandler.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h
```

```
$(BLD_PATH)tmsScEntry.Obj:\
    $(SRC_PATH)tmsScEntry.Asm
```

```
$(BLD_PATH)tmsScInfo.Obj:\
    $(SRC_PATH)tmsScInfo.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h\
    $(SRC_PATH)syscom.h
```

```
$(BLD_PATH)tmsScInst.Obj:\
    $(SRC_PATH)tmsScInst.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h
```

```
$(BLD_PATH)tmsSystem.Obj:\
    $(SRC_PATH)tmsSystem.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h
```

```
$(BLD_PATH)tmsString.Obj:\
    $(SRC_PATH)tmsString.Asm\
    $(SRC_PATH)string.mdf
$(CTOS_ASM)
```

```
$(BLD_PATH)tmsTaskInfo.Obj:\
    $(SRC_PATH)tmsTaskInfo.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErcs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h
```

```

#-----
#
#   tmsNull.run
#
$(BLD_PATH)tmsNull.run::\
    $(TMSNULL_ENTRY_OBJ)\
    $(TMSNULL_OBJ)\
    $(TMSNULL_LIBRARIES)
!IFDEF SYNTAX
!   IFDEF DEBUG
!       IFDEF BROWSER
            $(TRASH_OLD_BSC)
            $(BSC)\n $(BFLAGS) @<<
$(TMSNULL_BSC)
<<
        \g
!       ENDIF
!   ENDIF
            $(TRASH_OLD_RUN)
            $(LK)\n @<<
$(TMSNULL_ENTRY_OBJ)
$(TMSNULL_OBJ)
<<
        \n $(RUN_OUTPUT)\n $(MAP_OUTPUT)\n $(PUBLICS)\n
$(LINENUMBERS)\n\
            $(TMSNULL_STACK)\n $(TMSNULL_MAXARRAY)\n
$(TMSNULL_MINARRAY)\n\
            $(TMSNULL_RUNMODE)\n '$(VERSION)'\n @<<
$(TMSNULL_LIBRARIES)
none
<<
        \n $(DSALLOC)\n $(SYM_OUTPUT)\n $(COPYRIGHT)\n
$(FILETOAPPEND)\n\
            $(CONFIG_FILE)\n $(SOURCEDEBUGGER)\g
!ENDIF

$(BLD_PATH)tmsNullMain.Obj:\
    $(SRC_PATH)tmsNullMain.c

```

Sample NMAKE Description Files

```
#-----
#
#   tmsRequest.sys
#
$(BLD_PATH)tmsRequest.sys:\
    $(SRC_PATH)tmsRequest.txt
    $(TRASH_OLD_SYS)
    $(MRS)\n $(TXT_INPUT)\n $(SYS_OUTPUT)\n $(ERR_OUTPUT)\n
'$ (VERSION)' \g
    $(VID)\n $(PAUSE)\g
    $(CAT)\n $(ERR_OUTPUT)\g
    $(VID)\g

#-----
#
#   tmsAPI.lib
#
$(BLD_PATH)tmsAPI.lib:\
    $(TMS_API_OBJ)
!IFDEF SYNTAX
    $(TRASH_OLD_LIB)
    $(LIB)\n @<<
$(LIB_OUTPUT)
+$(TMS_API_OBJ);
<<
    \g
    $(APPEND)\n $(LEGALESE_TXT)\n $(LIB_OUTPUT)\g
    $(APPEND)\n [Kbd]\n
$(LIB_OUTPUT)\g\n$(LIBVER_LIB)\n$(LIBVER_VER)\n\ f
!ENDIF

$(BLD_PATH)tmsRqLabl.Obj:\
    $(SRC_PATH)tmsRqLabl.Asm\
    $(SRC_PATH)rqLabl.mdf
    $(CTOS_ASM)

$(BLD_PATH)tmsPipeAPI.Obj:\
    $(SRC_PATH)tmsPipeAPI.c\
    $(TMS_COMMON_H)\
    $(SRC_PATH)ctosErCs.h\
    $(SRC_PATH)ctosLib.h\
    $(SRC_PATH)ctosTypes.h
```

```
$(BLD_PATH)tmsPipeAPI2.Obj:\
    $(SRC_PATH)tmsPipeAPI2.Asm

#-----
#
#   evsAPI.lib
#
$(BLD_PATH)evsAPI.lib: \
    $(EVS_API_OBJ)
!IFDEF SYNTAX
    $(TRASH_OLD_LIB)
    $(LIB) \n @<<
$(LIB_OUTPUT)
+$(EVS_API_OBJ);
<<

    \g
    $(APPEND) \n $(LEGALESE_TXT) \n $(LIB_OUTPUT) \g
    $(APPEND) \n [Kbd] \n
$(LIB_OUTPUT) \n \g \n $(LIBVER_LIB) \n $(LIBVER_VER) \n \f
!ENDIF

$(BLD_PATH)evsRqLabl.Obj:\
    $(SRC_PATH)evsRqLabl.Asm\
    $(SRC_PATH)rqLabl.mdf
$(CTOS_ASM)

$(BLD_PATH)evsRqLablAlt.Obj:\
    $(SRC_PATH)evsRqLablAlt.Asm\
    $(SRC_PATH)evsRqLabl.Asm\
    $(SRC_PATH)rqLabl.mdf
$(CTOS_ASM)
```

This Page is Blank
Do Not Print

Glossary

This glossary defines terms used in this book, and CTOS-specific terms that programmer's commonly encounter.

A

ANSI Standard.

American National Standards Institute Standard for the C Language.

application partition.

A section of user memory reserved to execute applications. A workstation can have multiple partitions with concurrently running applications.

archive file.

A file created by one of the backup commands (for example, Backup Volume, Selective Backup, or Tape Backup). Archive files store the contents of other files in a compressed format.

archive media.

Floppy diskettes, QIC tapes, or half-inch tapes used to store archive files.

arithmetic operators.

Symbols used in arithmetic expressions that indicate the operation to perform. The standard operators are add (+), subtract (-), multiply (*), and divide (/).

array declarator.

A declarator followed by square brackets, possibly enclosing an integral constant expression. If no expression appears, the array has unknown size; otherwise, the expression is the number of elements in the array.

ASCII.

An acronym for the American Standard Code for Information Interchange. ASCII defines the character set codes used for information exchange between equipment. ASCII commonly denotes a file containing unformatted text.

_asm block.

A code block that starts with the `_asm` keyword, followed by an assembly instruction or group of instructions, possibly enclosed in braces.

Assembly Language.

The lowest-level language you can use to write CTOS programs. You can use CL to convert assembly files into CTOS object modules.

B**Bind.**

The Executive command that activates the Linker to create a Version 6 run file. Also the act of creating a Version 6 run file. Version 6 run files are required for protected mode compatibility.

BNet II.

Communications software program allows users to access the system files and other resources in remote locations. It includes utilities for data transfer and network administration.

boot.

The operation that starts a system when powered on or reset. The boot operation runs self-diagnostic tests, loads the operating system, and reads system files.

braces ({}).

The characters used to enclose a network node name in a file specification or path setting.

BTOS II.

A high performance Unisys operating system for B24, B26, B27, B28, B38, and B39 workstations and shared resource processors. See also CTOS.

buffer.

An area of memory used for temporary storage of input and output data.

byte.

A unit of computer information, usually composed of eight bits, and often representing one character.

C**C.**

A high-level language used to write CTOS programs. You can use the C compiler to convert the programs into object modules.

calling convention.

The way a language implements a call. The choice of calling convention affects the machine instructions that a compiler generates to execute and return from a function, procedure, or subroutine call; and the order in which the function arguments are passed (right to left or left to right).

case sensitive.

A parameter, language, or command that requires text to match a certain pattern exactly, including all uppercase and lowercase letters. Most CTOS commands are not case sensitive; that is, the operating system interprets COPY and copy the same way.

cast.

An abstract type enclosed in parentheses. Casts can appear after a sizeof keyword, or be used as a unary operator to convert the operand expression to the named type. Both the operand and the cast must have scalar type.

CL.

The compiler command supplied with CTOS Microsoft C.

client.

A general term for a user, software application, or computer that requires the services, data, or processing of another application or computer.

client process.

A process that requests a system service. All processes, even built-in operating system processes, can be client processes, since all processes can request system services.

cluster.

One or more workstations connected to a shared resource processor or server workstation.

cluster configuration.

A local resource-sharing network consisting of a server connected to cluster workstations. Each workstation has one high-speed RS-422 channel that enables intercluster communication. Large clusters with a shared resource processor may require multiple RS-422 channels. See also cluster workstation, server, and server workstation.

cluster workstation.

A workstation connected to a server workstation in a cluster configuration. A cluster workstation optionally has local file storage. See also cluster configuration and server.

COBOL (COmmon Business Oriented Language).

One of the high-level languages you can use to write CTOS programs. You can use the COBOL Compiler to convert source code into object modules.

code listing.

A readable display of compiled code.

code segment.

A variable-length (up to 64K) logical entity consisting of re-entrant code, and containing one or more complete procedures.

COMDEF OMF record.

COMmunal DEFinition, Object Module Format record that CTOS Microsoft C emits to the object file for uninitialized global variables.

command file.

A file the Executive reads to display command names, Executive command forms, and Help descriptions. The Executive associates a run file and command case with each command.

compiler.

A program that translates high-level language programs into object modules (machine-readable code).

configuration file.

A file that contains parameter values for a software product or hardware device. A configuration file specifies data used by utilities, applications, and the operating system.

Context Manager.

A software program that divides memory into multiple partitions in which you can start and concurrently run applications and utilities.

corrupted volume.

An initialized disk no longer recognized by the operating system.

CTOS.

A high performance Unisys operating system for B24, B26, B27, B28, B38, and B39 workstations and shared resource processors. CTOS is also an umbrella term encompassing all varieties of the BTOS, CTOS, and CTOS/XE operating systems.

CTOS.Lib.

Part of the Language Development software. It is a library of object modules that provide operating system run-time support.

D

default data segment.

See DGroup.

default value.

A value used by commands when optional fields are blank.

denormalized numbers.

Nonzero floating-point numbers with reserved exponent values in which the most-significant bit of the mantissa is zero.

dependency tree.

NMAKE's system of building files in the order needed to update the original target. NMAKE never builds targets until it updates (recompiles) all changed files.

dependent.

A file used to create a target, such as a C source file. Dependents are listed in makefiles.

description block.

Code that tells NMAKE how to build a project's target files. A description block specifies targets, dependents, and commands.

description file.

A description block plus comments, macros, inference rules, and directives, found in a makefile.

DGroup.

DGroup usually includes data, constant, and stack segments. It is also referred to as the default data segment.

directory specification.

The identifier for a directory. It consists of a node name, volume name, and a directory name.

distributed processing.

Using multiple processors to achieve a single result.

dynamically-installed system service

A program that loads as an application program, and then uses the CTOS ConvertToSys operation to convert itself into a system service. Once installed, a dynamically-installed system service has the same capabilities as a system service that was linked with the System Image during system build. A dynamically-installed system service uses operating system operations to identify the request codes that it services, specify its execution priority, establish its interrupt handlers, and so forth.

E

Envelope function.

For CTOS Microsoft 6.0, contains code to save and restore the SI and DI registers and clear the direction flag. You must link envelope function objects with your program in addition to the currently linked libraries. With CTOS Microsoft C 6.1, the `_plm` keyword generally makes these functions unnecessary.

erc.

See status code.

error code.

See status code.

error message.

A message that appears when an error occurs in an operating system, system service, or application. Problems that occur during bootstrap may appear as keyboard lights and audible beeps.

event.

Any external stimulus that causes an action to occur. The stimulus can be generated by users (keystrokes or mouse clicks), applications (unexpected messages), or operating systems (interval lapses in timer management).

Executive.

The CTOS command interpreter. The Executive is an interactive application program that requests the operating system to load programs that execute commands entered by the user. The Executive is usually loaded from the [Sys]<Sys>Exec.run file.

expansion.

The physical replacement of wild card characters for the actual characters they match.

exponent.

Second part of a floating-point number that contains the order of magnitude of the number. See also Mantissa.

expressions.

In a program, a combination of various constants, variables, operators, and parentheses used to perform a computation.

external reference.

A reference from one object module to variables and entry points of other object modules.

F

fh.

An acronym for file handle.

field.

An area that accepts parameter values in a command form.

file access methods.

Object module procedures located in the standard CTOS library. They provide buffering, and use the asynchronous input/output capabilities of the file management system to overlap input/output and computation.

file handle.

A 16-bit integer that uniquely identifies an open file. It is returned by the CTOS OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

file name.

A unique name that specifies a file. It can contain a maximum of 50 alphanumeric characters, including uppercase and lowercase letters, periods, hyphens, and right angle brackets. A file name is the fourth element of a full file specification.

file specification.

A unique identifier that contains the name of a file, as well as its volume, directory, and node location. A complete file specification has the form
(Node)[Vol]<Directory>FileName^password.

file system.

The data and control structures stored on an accessible disk device.

flag.

An indicator that denotes the existence of a particular condition. It can have a value of TRUE or FALSE.

full file specification.

The node name, volume name, directory name, file name, and optional password for a file.

function declarator.

A declarator followed by an optional set of language modifiers and a pair of parentheses, possibly empty.

G**general protection fault.**

A system fault generated by breaking any of the Protected mode protection rules. General protection faults occur when a process attempts to access memory that is not allocated to it.

group.

A named collection of Linker segments that the CTOS loader addresses at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

I**identifier.**

A sequence of uppercase and lowercase letters, digits, and the underscore character. An identifier must begin with a letter or the underscore. It can be any length, but the first 32 characters are significant only.

IEEE.

Institute of Electrical and Electronics Engineers.

inference rules.

Templates that NMAKE uses to create files with a given extension. Inference rules provide a convenient shorthand for common operations.

inherited macros.

Macros NMAKE creates that are equivalent to every current environment variable. These macros inherit the names and values of the corresponding environment variables.

initialization.

A process performed at the beginning of a program to ensure that all indicators and constants are set to prescribed conditions and values before you use the program.

interactive.

A program that users can manipulate.

J

Job Control Language (JCL).

A programming language processed by the Batch facility.

K

kernel.

The most primitive and the most powerful component of the operating system. The Kernel provides interprocess communication (IPC) primitives, and schedules process execution. It executes with a higher priority than any process, but it is not itself a process. See also process.

kilobyte.

1024 bytes of data or storage space.

L

Language Development.

The CTOS Language Development software that provides the Linker, CTOS Librarian, and CT Assembler programs (Bind, Librarian, and Assemble Executive commands).

.lib.

The standard file name suffix for library files.

library.

A collection of object modules (complete routines or subroutines) that can be linked into run files.

library file.

A file that contains one or more object modules. The file name normally includes the suffix .lib.

Link.

The Executive command that displays the Linker command form for combining a list of object modules into a version 4 run file. Also generically refers to the act of creating a run file.

linker.

A program that combines object modules into run files.

linker segment.

A single entity consisting of all segment elements with the same segment name.

loadable request file.

A binary file that contains request definitions for system services. The loadable request file merges new requests with the requests in Request.sys during installation of system services. When bootstrapped, the operating system reads Request.sys, loads it into memory, and adds the requests to the basic request routing table.

local context.

A Help file in which HelpMake embeds an encoded cross-reference that has meaning only within the current context (topic).

local file system.

The system software that manages disk access requests on cluster workstations with hard disks. The local file system allows cluster workstations to access files on local hard disks, as well as files on server hard disks. The filter process of the local file system intercepts each file access request and directs it to the local file system or to the server.

logical file address (lfa).

A 32-bit unsigned integer used to locate a particular sector of a file. An lfa specifies the byte position within a file, and is the number (offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0. For example, the lfa of the third sector of a file is 1024. An lfa must be on a sector boundary and is thus a multiple of 512.

M**.Map.**

.Map is the standard file name suffix for Linker list files.

mantissa.

First part of a floating-point number that contains the value of the number. See also Exponent.

map file.

The Linker list file (suffix .Map) that contains an entry for each Linker segment, identifying the segment relative address and length in the memory image. You can direct the Linker to list public symbols and line numbers.

MCBIND

One of two command forms supplied with CTOS Microsoft C that you can use to invoke the CTOS Linker. See also MLINK.

megabyte (MB).

1,048,576 bytes of data or storage space.

message file.

A binary file containing the screen prompts and messages displayed by an application.

MLINK

One of two command forms supplied with CTOS Microsoft C that you can use to invoke the CTOS Linker. MLINK executes automatically after successful compiles. See also MCBIND.

N

naming convention.

The way a compiler alters a routine name before placing it in the object file. You can choose between C and Pascal naming conventions to ensure that the names in the calling program agree with those in the called program.

NAN.

Acronym for Not A Number.

NCP.

See numeric coprocessor.

network node.

A server connected to BNet II or CT-Net. Cluster workstations connected to a node can communicate with other network nodes. Some releases of BNet II also allow cluster workstations to be network nodes.

NMAKE.

The program maintenance utility that simplifies project management by determining which project files depend on others. NMAKE can automatically update your project when any project file has changed.

numeric coprocessor.

An Intel 8087, 80287, or 80387 chip.

O**.obj.**

The standard file name suffix for object module files.

object module.

A file that results from a single compiler or assembler function. You can link the object module with other object modules into executable run files.

object module procedure.

A procedure supplied as part of an object module file. It is linked with the user-written object modules of an application program and is not supplied as part of the System Image. Most application programs require only a subset of these procedures. When the application program is linked, the desired procedures are linked together in the run file of the application.

optimization.

What the compiler does when it rewrites parts of a program to make it optimally efficient in ways that are not apparent at the source level.

output.

Data delivered from a program to a file or device.

P**parameter.**

A variable or constant transferred to and from a subroutine or program, affecting program execution.

parameter value.

An element of information supplied in a command form, procedural call, or configuration file.

partition.

A defined portion of workstation memory. There are two types of partitions: system and application. Each application runs in a separate partition. A partition management program such as Context Manager allows multiple application programs to execute simultaneously, since each program restricts instructions and data to its assigned memory area.

partition management.

A program that provides operations for creating, managing, and removing application partitions. Partition management permits concurrent execution of multiple application programs, each in its own partition. See also partition managing program.

partition managing program.

A program, such as Context Manager, which uses partition management operations to manage multiple application partitions in memory. Each partition can contain an executing application program. See also partition management.

Pascal.

One of the high-level languages you can use to write CTOS programs. The Pascal Compiler converts pascal source programs into CTOS object modules.

physical address.

An address relative to memory location 0. It does not specify a segment base.

pointer.

An address that specifies a storage location for data.

pointer declarator.

A declaration beginning with an asterisk (*).

processor.

The central processing unit (CPU), memory, and associated circuitry. It interprets and executes instructions.

program.

Executable code, data, and one or more processes. The code and data can be unique to the program or shared with other programs. A program is created by translating source programs into object modules. The Linker then links the modules together, creating a run file stored on disk. When requested by a currently active program, the operating system reads the run file into the application partition, relocates intersegment references, and schedules it for execution. The new run file can coexist with or replace other run files.

prompt.

A message that guides you through certain procedures. For example, when you use the FINISH command to finish a session, a prompt appears that tells you to confirm the command.

protected mode.

A processor mode in which application programs can access all available free memory above the first megabyte up to the maximum allowed by the processor and the hardware. See also general protection fault.

pseudotarget.

A target name that allows you to build groups of files or execute a group of commands.

PUBDEF record.

The PUBLIC DEFINITION record that CTOS Microsoft C emits to the object file for an initialized global variable.

public procedure.

A procedure that can be referenced from a module other than the defining module.

public installation.

An Installation Manager option that installs software on the server. Local file systems can later install the software from the server. In addition, users can delete some system files from the local disk and transparently access the corresponding publicly installed files on the server.

public symbol.

An identifier associated with a public variable, a public value, or a public procedure.

public value.

A value that can be referenced by a module other than the defining module.

public variable.

A variable that has a public address. Thus, a module other than the defining module can reference the address.

Q

QIC tape.

See quarter-inch cartridge tape.

quarter-inch cartridge (QIC) tape.

A magnetic tape storage medium that uses quarter-inch-wide tape contained in a hard plastic cartridge.

R

RAM.

An acronym for Random Access Memory.

random access memory (RAM).

The onboard processor memory where data can be written and read nonsequentially.

real mode.

A processor mode in which application programs can access memory within the first megabyte only.

reboot.

The action that resets hardware, and then reloads and executes the operating system. The operating system reboots automatically when a workstation is reset or turned on after a power failure, and when executing the BOOTSTRAP command.

release medium.

The floppy disk on which Unisys release software is distributed.

relocation.

The CTOS Loader relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file.

relocation directory.

An array of locators that the CTOS Loader uses to relocate the task image.

request.

A request by a client for an operation to be performed by the operating system or by a system service process.

request-based.

An operation that uses the request procedural interface. See also request procedural interface.

request block.

A block of memory, provided by a client, which contains a special type of message that is formatted according to specific conventions, and is used by all Inter-Process Communications to the operating system. An application program (or the operating system) sends a request block to the operating system to request that a particular operation be performed.

request code.

A 16-bit value that uniquely identifies a system service. The request code routes a request to the appropriate system service process, and specifies the currently requested process.

request file.

See loadable request file.

request procedural interface.

A routine within the operating system that is executed when a program calls a request-based operation. The routine builds a request block message and calls the Request primitive, while the calling program is placed in the waiting state at its default response exchange for the system service to respond. The request procedural interface is compatible with high-level languages such as C and Pascal, as well as assembly language. See also request and request-based.

RS-422.

A high-speed communications standard for interfacing serial data transmission between peripherals, systems, and modems. The shared resource processor and server workstations use the RS-422 interface to link cluster workstations.

.run.

The standard file name suffix for run files.

run file.

An executable CTOS program. It is created by the Linker and contains object modules linked together into code and data segments.

run statement.

A line of text that you can include in a JCL file for execution by Batch on a processor.

S

sector.

The smallest addressable portion of a track on a hard or floppy disk. It contains 512 bytes.

segment.

A contiguous area of memory that contains code or data.

segmented address.

An address that specifies both a segment base and an offset.

segment registers.

This manual refers to four segment registers: the Code Segment (CS) register, the Data Segment (DS) register, the Extra Segment (ES) register, and the Stack Segment (SS) register.

sequence point.

A point in the syntax of the C language at which all side effects of an expression or series of expressions have been completed. The statement operator (;) is such a point.

server.

A workstation or shared resource processor to which cluster workstations are attached. The server provides file system, queue management facility, and other services to all cluster workstations.

server workstation.

A server that supports its own interactive programs.

shared resource processor.

A multiprocessor that permits workstations in a cluster environment to share resources. It consists of various combinations of the GP, CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by the individual processor boards can be shared among all others. See also XE 520 and XE 530.

shift operator.

An operator that shifts the left bits of the integral quantity to the left (<<) or right (>>) by the amount given in the right-hand operand. A shift operator must have operands of integral type.

short-lived memory.

The memory area in an application partition. When CTOS loads a task, it allocates short-lived memory to contain the task code and data. A client process can also load short-lived memory in its own partition.

sign extension.

The propagation of the sign bit to fill unoccupied space when promoting to a more significant type, or when performing bitwise right-shift operations.

size.

The number of bytes a data item or structure contains, unless the context indicates a different unit of measurement.

source code.

The text of a computer program before it is compiled and linked to form an executable program.

stack.

A region of program memory used for temporary data and addressed using the stack segment (SS) and stack pointer (SP) registers.

stack frame.

A region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

stack pointer.

(SP) Indicates the top of a stack. The stack pointer is stored in the registers SS:SP.

Standard Software.

A set of programs, configuration files, and commands that are required to configure the system and perform basic operations.

statement

An executable instruction. Each statement is executed in order within a statement list unless otherwise directed by control-flow statements.

status code.

A decimal or hexadecimal number that reports the success or failure of the requested operation. A system service process stores a status code in a request block, where the client process examines it. A nonzero status usually indicates an error condition. Status code is also referred to as error code and etc.

style ID.

A six-character alphanumeric identifier for a specific Unisys product. For example, B39-CPU identifies a B39 workstation.

submit facility.

A utility that interprets a sequence of characters from a file as though they were characters typed at the keyboard. Submit files allow the convenient repetition of command sequences. See also submit file.

submit file.

A text file used by the system input process. It contains the same sequence of characters that would be typed from the keyboard. It can also contain conditional branching, variables, and video controls. When applications or requests activate submit files, file characters are returned in place of keyboard characters. See also system input process.

subparameter.

A user-supplied string in a parameter field of an Executive command form. You separate subparameters with a space.

.Sym.

.Sym is the standard file name suffix for the CTOS Linker's symbol file.

symbol.

An alphanumeric or other character, such as underscore, period, or dollar sign.

symbol file.

The Linker symbol file (suffix .Sym) that contains a list of all public symbols.

synchronous operation.

A protocol that requires a response at an exact time. Also a special type of synchronization using the CTOS request mechanism. For example, a program can issue the Wait primitive to wait at an exchange so it can synchronize its operation with a system service's response.

Sys.

An abbreviation for System. It is used in file names to denote files that are necessary for the workstation or shared resource processor to boot and function properly. It is also used as the name for the volume and directory that contain the operating system configuration.

system administrator.

The person responsible for planning, generating, extending, and controlling the use of the operating system and system services to improve the overall productivity of the installation.

system crash.

An abnormal condition from which the system cannot recover. After a crash, the system usually freezes or reboots automatically.

system disk.

The system disk is the disk on which the workstation server or shared resource processor system software is installed.

system error log.

A file containing information about many types of hardware and software errors. You can view or print it with the PLog command.

system input process.

A CTOS process that permits a sequence of characters from a submit file to be interpreted as though they were characters typed at the keyboard. Submit files allow the convenient automation of command sequences. See also submit facility and submit file.

system service.

A program or subprogram that expands the capabilities of the operating system. It manages or provides access to a resource. A resource may be hardware, such as a mouse or printer, or other system services.

T

target.

Listed in a makefile, a file that you want to create, such as an executable file.

task.

Executable code, data, and one or more processes.

task image.

A program stored in a run file that contains code segments and/or static data segments.

text file.

A file that contains bytes that represent printable characters or control characters (such as tab, newline, and so on).

U

user configuration file.

A file containing a user profile. See also user file.

user file.

A file in the system directory that identifies the user and specifies the environment the system activates after the user signs on.

user name.

The name a user signs on with. The file name of a user file, in the form [Sys]<Sys>UserName.user, defines the user name.

V

version 4 run file

A real-mode run file, one that cannot run above the 1 mb barrier. Version 4 run files are not compatible with protected mode.

version 6 run file

A protected-mode run file, one that can run above the 1 mb barrier.

W

wild card character.

A character that represents one or more unspecified characters. CTOS has two wild card characters: an asterisk (*) and a question mark (?). The asterisk represents any string of characters; the question mark represents individual characters. For some operations, you can use wild card characters in file specifications. The system then tries to match the portion of the name that appears before or after the wild card character and performs the requested operation for each matched file name. For example, *.ind would expand to all files that end in .ind.

X

XE 520.

A real-mode server multiprocessor system that permits workstations in a cluster environment to share resources. It consists of various combinations of the CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by the individual processor boards can be shared among all the others.

XE 530.

A protected-mode server multiprocessor system that permits workstations in a cluster environment to share resources. It consists of various combinations of the GP, CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system.

Index

- \ par, 11-16
 - \f escape sequence, 10-7
 - \g escape sequence, 10-7
 - \n escape sequence, 10-7
- _asm blocks
 - defining _asm blocks as
 - C, 8-18
 - referring to C variables, 8-11
 - tags not recognized in, 8-10
 - using assembly language
 - in, 8-4
- _asm keyword
 - defined, 8-2
- _beginthread function, 16-6
- _cdecl keyword, 6-29
- _cdecl prototypes, 5-36
- _ctoserrno, A-5
- _endthread function, 16-7
- _export keyword, 16-18
- _fastcall, 6-29
- _fastcall calling convention
 - argument-passing
 - convention, 6-31
 - function-naming
 - convention, 6-33
 - register preservation
 - requirement, 6-32
 - stack adjustment
 - convention, 6-32
- _fortran keyword
 - using, 14-13
- _huge memory support for real
 - mode, 5-21
- _loads keyword, 16-19
- _pascal keyword, 6-30
 - using, 14-13
- _plm keyword, 6-30
 - limitations, 5-37
 - using, 5-36
 - what it does, 14-14
- !CMDSWITCHES directive, 10-23
- !ELSE directive, 10-23
- !ENDIF directive, 10-23
- !ERROR directive, 10-23
- !IF directive, 10-23
- !IFDEF directive, 10-24
- !IFNDEF directive, 10-24
- !INCLUDE directive, 10-24
- !UNDEF directive, 10-24
- .category dot command, 11-24
- .command dot command, 11-24
- .comment dot command, 11-24
- .context dot command, 11-24
- .end dot command, 11-24
- .freeze dot command, 11-24
- .IGNORE pseudotarget, 10-27
- .length dot command, 11-24
- .list dot command, 11-25
- .mark dot command, 11-25
- .next dot command, 11-25
- .paste dot command, 11-25
- .popup dot command, 11-25
- .previous dot command, 11-25
- .ref dot command, 11-25
- .SILENT pseudotarget, 10-27
- .SUFFIXES pseudotarget, 10-27
- .topic dot command, 11-25
- /FPa floating-point option, 9-13
- /FPc floating-point option, 9-11
- /FPc87 floating-point option, 9-12
- /FPi floating-point compiler option, 9-10
- /FPi87 floating-point option, 9-11

- /G0 optimization option, 6-21
- /G1 optimization option, 6-21
- /G2 optimization option, 6-21
- /Gc optimization option, 6-30
- /Gd optimization option, 6-29
- /Gr optimization option, 6-30
- /Gs optimization option, 6-18
- /Gt optimization option, 7-26
- /ND optimization option, 7-27
- /Oa optimization option, 6-12
- /Oc optimization option, 6-20
- /Od optimization option, 6-5
- /Oe optimization option, 6-18
- /Og optimization option, 6-20
- /Oi optimization option, 6-8
- /Ol optimization option, 6-15
- /On optimization option, 6-16
- /Op optimization option, 6-20
- /Os optimization option, 6-8
- /Ot optimization option, 6-8
- /Ow optimization option, 6-12
- /Ox optimization option, 6-22
- /Oz optimization option, 6-16
- 80186 processor, 6-21
- 80188 processor
 - instruction set, 6-21
 - operating systems, 6-22
 - using, 6-21
- 80286 processor, 6-21
- 80x87 coprocessors
 - register size, 6-20
- 87.LIB
 - defined, 5-23

A

- abort function, C-27
- Accessing parameters in
 - assembly, 14-27
- Addition operator (+), 10-25
- Address space
 - portability, 15-19
- Address variables
 - mixed-language programming, 14-41

Addresses

- passing with common blocks
 - in mixed languages, 14-42

Addressing

- based, 7-5
- ADR type, 14-41
- ADRMEM, 14-41
- ADS type, 14-41
- ADSMEM, 14-41
- Advisor, 4-1
- Advisor Help library APIs,
 - 11-34
- Aggressive optimizations
 - enabling, 6-16
- Alias, 6-12
- Aliasing
 - assuming no, 6-12
 - rules for nonvolatile variables, 6-12
 - troubleshooting, 6-14
- Allocating global registers, 6-18
- Allocating local data in
 - assembly, 14-24
- ALT key

- maps to COPY, 3-2, 12-1
- Alternate math libraries
 - xLIBFA.LIB, 5-24
- Alternate math package, 9-6
- Ampersand (&), 14-41
- ANSI standards and CTOS
 - implementation
 - arguments to main(), C-3
 - arrays, C-11
 - availability of registers, C-12
 - bits per character, C-5
 - case distinction, C-4
 - casting integers to
 - floating-point values, C-10
 - casting pointers, C-11
 - character sequences, C-18
 - character sets, C-5
 - character testing, C-21
 - characters, C-5

- ANSI standards and CTOS
 - implementation (continued)
 - converting multibyte
 - characters, C-7
 - declarators, C-15
 - demotion of integer values,
 - C-8
 - diagnostics, C-2
 - environment, C-2
 - environment names, C-28
 - floating-point math, C-10
 - identifiers, C-4
 - integers, C-7
 - interactive devices, C-3
 - largest array size, C-11
 - library functions, C-19
 - multibyte characters, C-5
 - pointer subtraction, C-12
 - pointers, C-11
 - pragmas, C-18
 - preprocessing directives,
 - C-16
 - qualifiers, C-15
 - range of char values, C-7
 - range of integer values, C-8
 - registers, C-12
 - remainders, C-9
 - right shifts, C-9
 - signed bitwise operations,
 - C-9
 - significant characters without
 - external linkage, C-4
 - statements, C-15
 - structures, C-13
 - translation, C-2
 - truncation of floating-point
 - values, C-11
 - unions, C-13
 - unrepresented character
 - constants, C-6
 - wide characters, C-6
- Append mode
 - file position in, C-23
- Applications
 - building, 5-1, 16-1
- argData, 12-21
- Argument types and register
 - candidates, 6-31
- Arithmetic
 - one's complement, 15-14
 - pointer arithmetic
 - huge versus far, 7-4
 - pointer in huge models, 7-9
 - two's complement, 15-14
- Arithmetic mode
 - processor, 15-14
- Arrays
 - ANSI standards and CTOS
 - implementation, C-11
 - automatic void with huge,
 - 7-8
 - declaring in mixed languages,
 - 14-38
 - displaying in CodeView,
 - 13-21
 - equivalent declarations
 - among languages, 14-39
 - indexing in mixed languages,
 - 14-38
 - limits on huge, 7-8
 - names always passed as
 - pointers, 14-13
 - static and huge, 7-8
 - using with int data types,
 - 15-4
- Assembly
 - accessing parameters, 14-27
 - allocating local data, 14-24
 - C calls to, 14-21
 - entering procedures, 14-24
 - exiting procedures, 14-31
 - preserving register values,
 - 14-25

- Assembly (continued)
 - returning values, 14-30
 - setting up procedures, 14-23
 - standard interface method, 14-23
 - writing procedures, 14-22
- assert function
 - diagnostic printed by, C-21
- astcall calling convention
 - return value convention, 6-32
- Asterisk (*), 10-7
- At sign (@), 6-33, 11-16
- atexit function, C-27
- Availability of registers
 - ANSI standards and CTOS implementation, C-12

B

- Back command button, 4-17
- Backslash (\), 10-6
- Based addressing, 7-5
- Based variables, 7-30
- Binary mode, 5-41
- Binary streams
 - number of null characters that can be appended, C-23
- Bit fields
 - alignment, C-14
 - alignment of data defined in, 15-14
 - in structures, 15-12
 - order of assignment in structures, 15-13
 - size and alignment in structures, 15-13
 - storage, C-14
- Bits per character
 - ANSI standard and CTOS implementation, C-5
- Bitwise AND operator (&), 10-25
- Bitwise operations
 - AND (&), 10-25
 - OR (|), 10-25
 - right shift, 15-24
 - signed, C-9
 - XOR (^), 10-25
- Bitwise OR operator (|), 10-25
- Bitwise XOR operator (^), 10-25
- Blank lines, C-23
- BP as frame pointer, 14-24
- Braces ({})
 - enclosing an `_asm` block, 8-3
 - enclosing text Choice symbol (|), viii
- Brackets
 - square ([]), 10-26
- Bracketted file names
 - including, C-16
- Bracketted text brackets ([]), viii
- Breakpoints
 - displaying in CodeView, 8-7
 - selecting and editing in CodeView, 13-13
 - setting and removing in CodeView, 13-13
 - setting values in CodeView, 13-15
 - using in CodeView, 13-17
- Browser
 - building files for the, 5-11
 - creating a debug build, 3-55
 - debugging with the, 3-57
 - overview, 1-7
 - starting, 3-13
 - using, 3-55, 3-59

- BTOS C functions
 - documented but not supported, A-3
 - summary of status in CTOS Microsoft C, A-22
- BTOS C math library
 - summary of function locations in CTOS Microsoft C, A-30
- Buffering files, C-23
- Building applications, 5-1
- Building DLLs with Microsoft C, 16-21
- Building run files, 5-1, 3-31
- Buttons
 - command, 4-17
- Bytes
 - CTOS Microsoft C ordering, 15-31
 - order in a word, 15-10
 - ordering for long types on various machines, 15-32
 - ordering for short types on various machines, 15-32

C

- C
 - data type naming
 - conventions, 14-34
 - using, 8-8
 - calling convention, 6-29
 - calls
 - to assembly language, 14-21
 - to FORTRAN, 14-15
 - to high-level languages, 14-12
 - to Pascal and PL/M, 14-18

- C functions
 - _asm blocks to C functions, 8-17
 - migrating to CTOS Microsoft C, A-2
 - run time library functions for thread control, 16-5
- C run-time library support of long double types, 9-5
- C stack frame example, 14-28
- C string format, 14-35
- C symbols
 - using in _asm blocks, 8-9
- Call to math coprocessor
 - compiler option, 9-12
- Calling a FORTRAN function
 - from C, 14-17
- Calling a FORTRAN subroutine
 - from C, 14-15
- Calling a function, 6-25
- Calling a Pascal or PL/M function from C, 14-20
- Calling a Pascal or PL/M procedure from C, 14-19
- Calling conventions
 - _fastcall, 6-31
 - affects programming three ways, 14-7
 - Assembly-language and C, 14-22
- calloc function, C-27
 - defaults for languages, 14-33
 - effects of, 14-8
 - FORTRAN and Pascal compared, 14-7
 - mixed-language programming, 14-7
 - stack adjustment, 6-32

Calls

- C to FORTRAN and Pascal, 14-12
 - from C to FORTRAN subroutines, 14-15
 - mixed-language protocol, 14-7
 - steps for executing a mixed-language call from C, 14-12
 - to Assembly, 14-21
 - to emulator compiler option, 9-11
 - to programs written in other languages, 14-1
 - to routines in different languages, 14-3
- Caret (^), 10-9
- Case distinction
- ANSI standard and CTOS implementation, C-4
 - in NMAKE macros, 10-15
- Case sensitivity
- portability issues, 15-26
- Case translation
- migration issues, A-7
 - portability issues, 15-21
- Casting
- pointer portability issues, 15-16
- Casting integers to floating-point values
- ANSI standard and CTOS implementation, C-10
- Casting pointers
- ANSI standards and CTOS implementation, C-11
- Char data type
- range of values, C-7
 - size, 15-3

Character

- ANSI standard and CTOS implementation, C-5
 - bits per, C-5
- Character classification
- portability issues, 15-20
- Character constants, C-6, C-16
- Character sequences
- ANSI standards and CTOS implementation, C-18
- Character sets
- ANSI standard and CTOS implementation, C-5
 - portability, 15-20
 - portability issues, 15-30
 - portability restrictions, 15-20
 - source and execution, C-6
- Character testing
- ANSI standard and CTOS implementation, C-21
- Characters
- case sensitivity and CL command line options, C-4
 - converting multibyte characters, C-7
 - multibyte, C-5
 - number of significant without external linkage, C-4
 - significance of case distinction, C-4
 - terminating newline, C-22
 - wide, C-6
- CHAR_BIT, 15-4
- CHAR_MAX, 15-4
- CHAR_MIN, 15-4
- Check_stack pragma, 6-18
- CL
- automatically invoking MASM, 5-3
 - command form, 5-2
 - command form summary, B-1

CL (continued)

- command line option notes, 5-3
- command line options, B-1
- command line syntax, 5-2
- controlling automatic linking, 5-9
- default memory model, 5-3
- global register allocation
 - option, 15-26
- how to use, 5-2
- overlay switches, 5-4
- overview, 1-2
- sample command form that creates a run file, 5-10
- sample command form with linker options, 5-21
- syntax errors, 5-5
- CLD instruction, 5-36
- Clock cycles of processor for calling sequence, 6-27
- clock function, C-30
- cmdTable, 12-16
- CodeView
 - advanced techniques, 13-29
 - calling functions, 13-30
 - changing data, 13-28
 - changing register values, 13-28
 - changing variables, 13-20
 - checking for undefined pointers, 13-30
 - command form summary, B-10
 - configuring, 2-14
 - conflict with CTOS Librarian, 3-63, 13-3
 - controlling program execution, 13-12
 - creating debug builds with MCBIND, 13-4

CodeView (continued)

- creating debug builds with MLINK, 13-4
- customizing from TOOLS.INI, 13-33
- dialog command size specifiers, B-18
- displaying arrays, 13-21
- displaying expressions in the Watch window, 13-20
- displaying memory, 13-25
- displaying processor registers, 13-27
- displaying structures, 13-21
- displaying variables in the Watch window, 13-19
- displaying in CodeView, 13-26
- executing programs
 - continuously, 13-12
- format specifiers, B-18
- handling register variables, 13-32
- modifying variables, registers, and memory, 13-27
- opening multiple source windows, 13-30
- overview, 1-6
- preparing a debug build, 3-62, 13-2
- print from, 13-31
- redirecting input and output, 13-32
- replaying a debug session, 13-28
- Run and Animate commands, 13-18
- selecting and editing breakpoints, 13-13

- CodeView (continued)
 - setting and removing
 - breakpoints, 13-13
 - setting breakpoint values, 13-15
 - setting command-line arguments, 13-29
 - single stepping through code, 13-18
 - starting, 13-6
 - starting and running from PWB, 3-64
- Step and Trace commands, 13-18
- system configuration and starting, 13-7
- using breakpoints, 13-17
- using History On, 13-29
- using the Quick Watch window, 13-24
- viewing and modifying program data, 13-19
- CodeView windows
 - adjusting, 13-11
 - entering commands in, 13-10
 - identifying active, 13-10
 - list of, 13-8
 - opening manually and automatically, 13-9
 - selecting, 13-10
- Coding conventions
 - migration issues, A-14
- Combined libraries
 - creating and naming, 5-25
 - creating with MLIB, 5-26
 - xLIBC7.LIB, 5-25
 - xLIBCA., 5-25
 - xLIBCE.LIB, 5-25
- COMDEF OMF record
 - versus PUBDEF record, A-20
- Command buttons, 4-17
- Command form summaries
 - CL, B-1
 - CodeView, B-10
 - Environment Service, B-19
 - ML, B-29
 - MLIB, B-28
 - NMAKE, B-38
 - PWB, B-40
 - PWBRMAKE, B-41
 - QuickHelp, B-43
- Command forms
 - CV, 13-7
 - CV Run, 13-7
 - HelpMake, 11-5
 - MCBIND, 5-20
 - MLIB, 5-26
 - NMAKE, 10-3
 - QuickHelp, 4-2
- Command line
 - controlling optimization from, 6-3
- Command line options
 - CL, B-1
 - HelpMake, 11-6
 - MLIB, B-28
 - MLINK, B-29
 - NMAKE, 10-28, B-38
 - PWBRMAKE, B-41
 - QuickHelp, B-43
- Command lines
 - indentation of NMAKE, 10-6
- Commands
 - .context, 11-2, 11-15
 - Browser, 3-59
 - CL, 5-2
 - CodeView, B-12
 - CV, 13-7
 - CV RUN, 13-7
 - HelpMake syntax, 11-5
 - MLINK, 5-7
 - NMAKE, 10-3

- Commands (continued)
 - QuickHelp, 4-2
 - QuickHelp dot, 11-23
- Comments
 - in a description file, 10-11
 - in-line assembly, 8-19
 - nested, 10-11, A-19
 - using old-style C in `_asm`
 - blocks, 8-18
- Common blocks
 - mixed-language
 - programming, 14-42
- Common subexpression elimination, 6-5
- Common subexpression optimization, 6-20
- Compile options, 16-19
- Compiler
 - how to use CL, 5-2
 - inserting leading underscores
 - with `_plm` and `_pascal`
 - keywords, 14-5
 - limits, D-2
 - limits and numerical ranges,
 - D-1, D-4
 - naming conventions, 14-4
 - numerical ranges for
 - constants defined in `LIMITS.H`, D-4
 - numerical values for
 - constants defined in `FLOAT.H`, D-6
 - optimization, 6-1
 - options and migration issues,
 - A-15
 - program limits at run time,
 - D-3
 - three methods for passing
 - parameters, 14-9
- Compiler assumptions
 - program portability, 15-22
- Compiler options
 - `/Ad`, 7-23
 - `/Au`, 7-24
 - `/Aw`, 7-24
 - `/GY` disabling `/Oc`, 5-4
 - `/GY` incompatibility with
 - intrinsic forms of `mem*` and `str*`, 5-5
 - `/r` disabling `/Oc`, 5-4
 - `/r` incompatibility with `/Oe` and `/Og`, 5-4
 - calls to emulator, 9-11
 - calls to math coprocessor,
 - 9-12
 - choosing for memory models,
 - 7-7
 - in-line emulator, 9-10
 - in-line math coprocessor,
 - 9-11
 - migration issues, A-15
 - setting in PWB, 3-35
 - summary of BTOS C and CTOS Microsoft C, A-16
 - use alternate math, 9-13
 - used for the Browser, 3-13
 - used to stop automatic
 - linking with `MLINK`, 5-3
 - using to migrate huge
 - memory models, A-11
- Compiling
 - choosing when compiling
 - mixed languages, 14-11
 - FORTRAN routines with the `$TRUNCATE`
 - metacommand enabled,
 - 14-5
 - mixed-language
 - programming, 14-11
 - overview, 1-2
 - PWB extensions, 12-29

- Compiling (continued)
 - with different languages, 14-1
 - COMPLEX*16 type, 14-35
 - COMPLEX*8 type, 14-35
 - Configuration file
 - Environment Service, 2-12
 - linker, 6-24
 - Constant propagation, 6-6
 - Constants
 - character, C-16
 - unrepresented character, C-6
 - Contents command button, 4-17
 - Context Manager
 - creating a QuickHelp partition, 4-25
 - Context prefixes, 11-17
 - Contexts
 - defined, 11-2
 - local, 11-16
 - Control characters
 - using as literals, 10-8
 - Conventions
 - manual, viii
 - mixed-language programming, 14-4
 - Pascal, 14-33
 - Converting header files, A-7
 - CPU registers
 - use of, 6-18
 - Creating run files
 - sample CL command form, 5-10
 - Cross references
 - defined, 11-2
 - hyperlinks, 11-2
 - implicit, 11-2
 - maps to CODE, 3-2, 12-1
 - QuickHelp, 11-27
 - CTOS III Applications
 - building, 16-1
 - CTOS Microsoft C
 - new features, 1-11
 - overview, 1-1
 - CTOS Microsoft Help File
 - Creation Utility (HelpMake), 11-1
 - CTOS Microsoft Macro Assembler (MASM)
 - advantages over in-line assembler, 14-21
 - versus using in-line assembly, 8-1
 - CTOS.Lib, 2-14
 - CTOS.Lib.h, 5-36, A-7
 - CTOSToolKit.Lib, 2-14
 - CURRENT.STS, 3-2, 3-31, 13-9
 - Customizing your
 - optimizations, 6-7
 - CV
 - command form, 13-7
 - CV RUN
 - command form, B-10
- ## D
- Dash (-), 11-6, C-25
 - Data
 - handling in mixed-language programming, 14-32
 - Data files
 - portability issues, 15-30
 - Data ranges defined in
 - LIMITS.H, D-4
 - Data threshold
 - setting, 7-26
 - Data type
 - char sizing, 15-3
 - declaring floating-point, 9-1
 - int sizing, 15-3
 - naming conventions for C, FORTRAN, and Pascal,

- Data type (continued)
 - 14-33
 - order and alignment issues
 - with structures and unions, 15-7
 - short int sizing, 15-3
 - size of basic, 15-2
 - variable size, 8-6
- Date
 - default, C-19
- Directives
 - data and operations, 8-5
 - even and align, 8-5
 - supported by in-line assembler, 8-5
- DBL_DIG, 15-6
- DBL_DIG, D-6
- DBL_MAX, 15-6
- DBL_MAX, 15-6
- DBL_MAX_10_EXP, 15-6
- DBL_MAX_EXP, 15-6
- DBL_MIN, D-6
- DBL_MIN_10_EXP, 15-6
- DBL_MIN_EXP, 15-6
- Dead-store elimination, 6-6
- Debug builds
 - advanced CodeView
 - techniques, 13-29
 - creating Browser, 3-13
 - creating for CodeView, 13-2
 - from PWB, 3-53
 - overview of CodeView
 - techniques, 13-11
 - problems to avoid in
 - assembly, 8-7
 - with error messages in PWB, 3-54
 - with the Browser, 3-13, 3-55, 3-57
- Declarators
 - ANSI standards and CTOS
 - implementation, C-15
- Decoding Help databases
 - decoded format Help files
 - 11-4
- Decoding options
 - HelpMake, 11-11
- Default optimizations, 6-5
- Demotion of integers
 - ANSI standard and CTOS
 - implementation, C-8
- Denormalized numbers
 - extending range of
 - floating-point numbers
 - with, 9-3
- Dependency line
 - NMAKE, 10-4
- Dependency tree, 10-6
- Dependents
 - defined, 10-2
- Description blocks
 - control characters as literals,
 - 10-8
 - dependency tree, 10-6
 - dependents, 10-5
 - escape sequences, 10-7
 - example, 10-5
 - extension search paths, 10-21
 - indentation, 10-6
 - NMAKE, 10-4
 - pseudotargets, 10-27
 - specifying a target in
 - multiple, 10-10
 - targets, 10-5
 - wildcards, 10-7
- Description command button, 4-22
- Description files
 - comments, 10-11
 - defined, 10-2
 - dependents, 10-5
 - description block example,
 - 10-5

Description files (continued)

- description blocks, 10-4
- directives, 10-23
- elements, 10-4
- escape sequences, 10-7
- example, creating a library, G-21
- example, creating a library version string, G-4
- example, creating a PWB extension, G-6
- example, creating a run file, G-12
- example, creating run files, libraries, and a system service, G-30
- extension search paths, 10-21
- how NMAKE interprets, 10-31
- listing targets in multiple description blocks, 10-10
- macros, 10-11
- NMAKE examples, G-1
- placing newline characters in, 10-9
- targets, 10-5
- user defined inference rules, 10-20
- using control characters as literals, 10-8
- wildcard characters, 10-7

Designing DLLs

- designing and writing, 16-13

DGroup

- controlling placement of near heap memory, 5-27
- placing near heap memory in, 5-28

Diagnostics

- ANSI standard and CTOS implementation, C-2

Dialog boxes, 4-15

Dialog commands

- CodeView, B-14

Directives

- !CMDSWITCHES, 10-23
- !ELSE, 10-23
- !ERROR, 10-23
- !IF, 10-23
- !IFDEF, 10-24
- !IFNDEF, 10-24
- !INCLUDE, 10-24
- !UNDEF, 10-24
- DB, 8-7
- EXTERN and assembly, 14-23
- for NMAKE description files, 10-23
- NMAKE, 10-23
- PUBLIC and assembly, 14-23
- defaults, 2-5

Directory name

- size, C-24

Disk space used by an executable file

- reducing, 6-24

Division operator (/), 10-25

DLLs

- and CTOS Microsoft C libraries, 16-10
- application programs, 16-10
- linking files needed, 16-26
- with static C run-time library functions, 16-21
- without C run time library functions, 16-12

- Domain errors, C-22
- Dot commands, 11-23
- Double colon (::), 10-10
- Double dollar sign (\$, 10-14
- Double float data type
 - size, 15-5
- Double float data type
 - support, 9-1
- DS allocation
 - migration, A-21
- Dynamic linking with CTOS III
 - DLL, 16-8

E

- EBADF constant, C-26
- EDOM error constant, C-22
- Effects of calling conventions,
 - 14-8
- EINVAL constant, C-26
- Ellipsis, viii
- EM.LIB
 - defined, 5-23
- Emulator package, 9-5
- Encoding options, 11-7
- End-of-file escape sequence,
 - 10-7
- ENOENT constant, C-28
- Entering the assembly
 - procedure, 14-24
- Envelope function
 - migrating to 6.1, 5-36
- Environment
 - ANSI standard and CTOS
 - implementation, C-2
- Environment differences
 - program portability issues,
 - 15-30
- Environment names
 - ANSI standard and CTOS
 - implementation, C-28

- Environment Service
 - command form summary,
 - B-19
 - commands, B-20
 - disabling CL and MLINK
 - variables for PWB, 3-2
 - disabling CL and MLINK
 - variables for PWB, 5-34
 - installing, 2-12
 - overview, 1-11
 - using, 5-33
 - variables, examples, and
 - associated commands, B-21
- Environment variables
 - CL description, B-22
 - descriptions, B-22
 - HELPPFILES description,
 - B-22
 - INCLUDE, C-16
 - INCLUDE description, B-22
 - INIT description, B-22
 - LIB description, B-22
 - M, B-22
 - MAKEFLAGS desc, B-22
 - MLINK, 5-11
 - NO87, 9-15
 - PATH description, B-22
 - QH description, B-22
 - recognized by QuickHelp,
 - B-43
 - recognized by CL, B-2
 - recognized by CodeView,
 - B-10
 - recognized by HelpMake,
 - B-23
 - recognized by MLIB, B-28
 - recognized by MLINK, B-30
 - recognized by NMAKE, B-38
 - recognized by P, B-40
 - TMP description, B-22

- EnvironmentConfig.Sys
 - sample, 2-12
 - syntax rules, 5-33
- Equality operator (), 10-25
- ERANGE macro, C-22
- errno, A-5
- errno values
 - supported, 5-41
- Error information
 - accessing in QuickHelp, 4-23
- Error messages
 - errno values supported, 5-41
 - interpreting in PWB, 3-54
 - manipulating, 5-40
- Errors
 - checking returns from
 - dynamic memory allocation routines, 15-19
 - domain, C-22
 - file position, C-26
 - message syntax, C-2
- Escape sequences
 - \f, \g, and \n for NMAKE
 - description files, 10-7
 - NMAKE, 10-7
- EVBackgroundOff
 - Config.Sys entry to make VGA work, 2-14
- EVEN Directives
 - ALIGN, 8-5
 - CodeView commands, B-12
 - dialog commands, B-14
- Example command button, 4-22
- Exclamation point (!), 10-23
- Executable file
 - reducing disk space used, 6-24
- Exiting assembly procedures, 14-31
- Exiting QuickHelp, 4-4
- Explicit cross references, 11-27

- Exponents
 - bias, 9-3
 - length of for floating-point data types, 9-2
 - storage, 9-3
- Expressions
 - displaying in CodeView, 13-20
 - evaluation order and portability, 15-27
 - SIZE, 8-9
- Extended keywords
 - summary of BTOS C and CTOS Microsoft C, A-13
- Extension search paths
 - NMAKE, 10-21
- Extensions
 - PWB extensions, 12-13
- External data
 - mixed-language programming, 14-40

F

- facility, A-21
- Far call optimization
 - enabling, 6-25
- Far heap memory
 - allocating, 5-31
 - sizing and allocating, 5-30
- Far pointers
 - how they work, 7-3
- FarCallTranslation, 6-25
 - benefits, 6-26
 - how it affects code, 6-26
- fdtofh function, A-4
- fgetpos function, C-26
- File access limits, C-24
- File buffering, C-23
- File compression, 11-7

- File handles
 - increasing the number of, 2-15
- File names
 - characters permitted, C-24
 - including bracketted, C-16
 - quoted, C-17
 - size, C-24
- File position errors, C-26
- File streams, A-5
- Files
 - deleting open, C-24
 - increasing the maximum number of open, 2-15
 - increasing the number of handles, 2-15
 - increasing the number of streams, 2-16
 - zero-length, C-24
- Flavor object modules
 - defined, 5-21
 - MCMEMORY.Obj, 5-32
 - MIN.Obj, 5-22
 - REALHUGE.Obj, 5-22
- Float data type
 - portability assumptions, 15-5
 - size, 15-5
 - support for, 9-1
- FLOAT.H
 - numerical values defined in, D-6
 - numerical values for defined constants, D-6
- Floating-point data types
 - declaring, 9-1
 - declaring functions that return, 9-4
 - declaring variables as, 9-2
 - differences between, 9-2
 - range, 9-3
- Floating-point math
 - alternate math package, 9-6
 - ANSI standard and CTOS implementation, C-10
 - compatibility between compiler options, 9-14
 - compiling and linking options, 9-7
 - controlling operations, 9-1
 - coprocessor package, 9-6
 - emulator package, 9-5
 - library considerations, 9-13
 - packages, 9-5
 - required for DLLs, 10-13
 - selecting operations, 9-7
 - summary of compiler options, 9-8
- Floating-point math libraries
 - 87.LIB, 9-5
 - EM.LIB, 9-5
 - xLIBFA.LIB, 9-11
 - xLIBFP.LIB, 9-9
- Floating-point numbers
 - extending range of with denormalized format, 9-3
- Floating-point results
 - achieving consistent, 6-20
- Floating-point routines
 - accessing, 9-7
- Floating-point values
 - casting integers to, C-10
 - effect of /Op on, 6-21
 - minimum and maximum, 9-3
 - truncation, C-11
 - underflow, C-22
- Floating-point variables
 - precision and storage compared, 9-4
 - promoting, 9-4

- FLT_DIG, 15-6
- FLT_DIG, D-6
- FLT_MAX, 15-6
- FLT_MAX, D-7
- FLT_MAX_10_EXP, 15-6
- FLT_MAX_EXP, 15-6
- FLT_MIN, 15-6
- FLT_MIN, D-7
- FLT_MIN_10_EXP, 15-6
- FLT_MIN_EXP, 15-7
- fMeta, 12-21
- fmod function, C-22
- Format specifiers
 - printf, E-1
 - scanf, E-1
- Formatting flags
 - QuickHelp, 11-26
- FORTRAN
 - accessing common blocks
 - directly, 14-43
 - C calls to, 14-15
 - calling a subroutine from C, 14-15
 - calling conventions, 14-7
 - data type naming
 - conventions, 14-34
 - indexing arrays, 14-38
 - naming conventions, 14-4
 - routines compiled with the \$TRUNCATE
 - metacommand enabled, 14-5
 - string format, 14-36
 - variable length strings not supported, 14-36
- FORTRAN/Pascal calling
 - convention, 6-30
- fprintf function, C-25
- Frame pointer, 14-24
- ftell function, C-26
- Function
 - not supported, 5-42

- Function calls
 - restrictions on using intrinsic forms, 6-11
- Functions
 - _fastcall, 6-32
 - abort, C-27
 - assert, C-21
 - atexit, C-27
 - BTOS C locations in CTOS
 - header files, A-22
 - calling FORTRAN from C, 14-17
 - calling from CodeView, 13-30
 - calling two ways, 6-25
 - character limits, A-15
 - clock, C-30
 - documented in BTOS C but not supported by CTOS Microsoft C, A-3
 - documented in BTOS C with functional substitutes in CTOS Microsoft C, A-4
 - envelope, 5-36
 - fgetpos, C-26
 - floating-point without true intrinsic forms, 6-10
 - fmod, C-22
 - fprintf, C-25
 - ftell, C-26
 - generating intrinsic, 6-8
 - intrinsic forms of mem* and str*, 5-5
 - intrinsic forms used when compiling with /Oi, 6-9
 - isalnum, C-21
 - isalnum, C-21
 - isctrl, C-21
 - islower, C-21
 - isprint, C-22

Functions (continued)

- isupper, C-22
- longjmp, 5-3
- messages generated by
 - perror, C-26
- migrating to
 - CTOS Microsoft C, A-2
- migration issues, A-22
- miscellaneous behaviors,
 - 5-42
- perror, 5-40
- portability issues, 15-28
- portable run-time, 15-20
- prototypes and migration,
 - A-14
- prototyping, 8-9
- rename, C-25
- returning values to assembly
 - code, 8-12
- Run-time library functions
- re-entrant, F-1
- rules for declaring near and
 - far, 7-18
- setjmp, 5-3
- strerror, C-28
- summary of BTOS C status,
 - A-22
- system, C-28
- that return floating-point
 - types, 9-4
- undocumented BTOS C with
 - work-arounds, A-5
- with a variable number of
 - arguments, 15-27

Function-calling conventions,
6-29

G

- GDT protected run files
 - creating with MLINK, 5-25
- Global register allocation, 6-18
- Global register allocation
 - CL option, 15-26
- GO escape sequence, 10-7
- Greater than operator (>),
 - 10-25

H

- H. contexts, 11-17
- Hardware assumptions
 - program portability, 15-2
- Header files
 - converting for migration, A-7
 - correlating BTOS C and
 - CTOS Microsoft C, A-8
 - migration, A-6
- Help
 - accessing error information,
 - 4-23
 - copying and pasting, 4-18
 - dialog boxes, 4-15
 - displaying keyword
 - information, 4-21
 - navigating through screens,
 - 4-15
 - printing, 4-24
 - structure, 4-5
 - using, 4-1
 - using hyperlinks, 4-17
 - using pull-down menus and
 - dialog boxes, 4-7

- Help database
 - content and format, 11-1
 - creating a, 11-13
 - using formats, 11-21
- Help delimiter (> >), 11-16
- Help file formats
 - ASCII, 11-22
 - minimally formatted ASCII, 11-4
 - QuickHelp, 11-4
 - QuickHelp components, 11-22
 - QuickHelp cross references, 11-27
 - QuickHelp dot commands, 11-23
 - QuickHelp formatting flags, 11-26
 - RTF, 11-22
 - supported, 11-3
- Help files
 - ASCII format, 11-4
 - assumptions about format and file structure, 11-15
 - contents, 11-2
 - context prefixes, 11-17
 - controlling text appearance with Helpmake format flags, 11-3
 - creating with Helpmake, 11-1
 - creating with two methods, 11-13
 - cross references, 11-2
 - database content and format, 11-1
 - decoding, 11-5
 - decoding options, 11-11
 - encoding, 11-5
- Help files (continued)
 - examples of how to create, 11-5
 - formats supported, 11-3
 - h. contexts, 11-17
 - hyperlinks, 11-19
 - local contexts, 11-16
 - manifest constants, 11-29
 - QuickHelp format, 11-4
 - RTF format, 11-22
 - structure of databases, 11-15
 - supported standard formats, 11-1
 - transferring between applications, 11-4
 - using the three text formats, 11-21
- Help text file structure, 11-15
- HelpMake
 - command examples, 11-5
 - command line options, 11-6
 - command form summary, B-23
 - command syntax, 11-5
 - creating Help files, 11-1
 - decoding options, 11-11
- High-level languages
 - C calls to, 14-12
- How languages push parameters onto the stack, 14-8
- Huffman compression, 11-7
- Huge memory model
 - migration, A-11
- Huge pointers
 - how they work, 7-4
- Hyperlink cross references, 11-2
- Hyperlinks
 - command buttons, 4-17
 - defined, 11-3
 - navigating through Help with, 4-17
 - using, 4-17

I

- Identifiers
 - ANSI standard and CTOS implementation, C-4
 - length, case, and portability, 15-25
 - naming convention in mixed-language programming, 14-4
- Implementation-defined behavior
 - ANSI related, C-1
- Implicit cross references, 11-2
- INCLUDE environment
 - variable, C-16
- Increasing the maximum
 - number of open files, 2-15
- Increasing the number of file handles, 2-15
- Increasing the number of streams, 2-16
- Index command button, 4-17
- Inequality operator (!), 10-25
- Inference rules, 10-20
- Initialization
 - and termination for DLLs, 16-14
 - migration issues, A-15
- Input redirection, 5-39
- Installation
 - and configuration, 2-11
 - default directories, 2-5
 - Environment Service, 2-12
 - OS requirements, 2-1
 - preliminary operations, 2-2
 - RAM requirements, 2-1
 - Task Management Service, 2-11
 - VGA notes, 2-14
- Instruction set
 - in-line assembly instruction set, 8-4
- Int data type
 - size, 15-3
 - size assumptions and portability, 15-4
- Integer values
 - range, C-8
- Integers
 - ANSI standard and CTOS implementation, C-7
 - casting to floating-point values, C-10
 - demotion, C-8
- Interactive devices
 - ANSI standard and CTOS implementation, C-3
- Intrinsic function calls
 - restrictions on using, 6-11
- Intrinsic functions
 - generating, 6-8
 - math, 6-10
- Intrinsic pragma, 6-10
- INT_MAX, 15-5
- INT_MAX, D-5
- INT_MIN, 15-5
- INT_MIN, D-5
- In-line assembler
 - advantages of in-line, 8-2
 - advantages over MASM, 14-22
 - in-line code features, 8-2
 - MASM advantages over, 14-21
 - uses of in-line, 8-2
 - writing in-line assembly, 8-12
- In-line emulator compiler
 - option, 9-10
- In-line files
 - NMAKE, 10-30

In-line instructions or calls,
 9-13
In-line math coprocessor
 instructions compiler option,
 9-11
isalnum function, C-21
isalpha function, C-21
isctrl function, C-21
islower function, C-21
isprint function, C-22
isupper function, C-22
Italicized text, viii

K

Keystrokes

- `_asm` before assembly
 instruction, 8-18
- `_based`, 7-12
- `_cdecl`, 6-29
- `_far` addressing, 7-12
- `_fastcall` Functions
- `_fortran`, 14-13
- `_fortran` and `maxparam`,
 14-16
- `_huge`, 7-8
- `_near` addressing, 7-12
- `_near` and `_far` with `_plm`,
 `_fortran`, and `_pascal`, 14-13
- `_pascal`, 6-30, 14-13
- `_plm`, 5-36, 6-30, 14-13
- `_plm` and `_pascal`, 14-5
- `_plm` effects, 14-14
- `_segment`, 7-32
- `_segname`, 7-32
- `_self`, 7-32

Keystrokes (continued)

- assigning in PWB, 12-4
- changing in PWB, 12-4
- disabling in PWB, 12-5
- examples of using `_plm`,
 `_pascal`, and `_fortran`, 14-14
- migration of language, A-12
- mixed-language naming
 conventions, 14-5
- nonportable, 15-31
- specific to CTOS Microsoft C,
 15-31
- summary of BTOS C and
 CTOS Microsoft C extended,
 A-13
- syntax rules and examples,
 14-14
- used with `_based` pointer
 declarations and
 statements, 7-32
- using `_far` in mixed
 languages, 14-11
- using `_plm`, `_fortran`, and
 `_pascal`, 14-12
- public and using `_fastcall`
 with mixed languages
 Compiler
 /Gr option and mixed
 languages, 14-9

L

Labels

- jumping to `_asm` blocks
- jumping to labels, 8-15

Language convention
 requirements, 14-4

- Language equivalents for
 - routine calls, 14-3
- Language keywords
 - migration, A-12
- Largest array size
 - ANSI standards and CTOS
 - implementation, C-11
- LDBL1_MIN, 15-7
- LDBL_DIG, 15-7
- LDBL_MAX, 15-7
- LDBL_MAX_10_EXP, 15-7
- LDBL_MAX_EXP, 15-7
- LDBL_MIN_10_EXP, 15-7
- LDBL_MIN_EXP, 15-7
- Left shift operator (<<), 10-25
- Less than operator (<), 10-25
- Less than or equal to operator (<=), 10-25
- LIBCEXT.LIB
 - defined, 5-23
- Librarian
 - overview, 1-10
- Libraries
 - 87.LIB, 5-23
 - combined, 5-22
 - conflicts between C and Pascal, 14-3
 - coprocessor, 5-23
 - creating and naming
 - combined, 5-25
 - CTOS, 2-14
 - CTOS.Lib, 5-24
 - default, 5-17
 - EM.LIB, 5-23
 - emulator, 5-23
 - LIBCEXT.LIB, 5-23
 - list of standard, 5-23
 - required for linking, 5-21
 - specifying standard, 5-22
 - unsupported functions, 5-42
 - xLIBCP.LIB, 5-23
 - xLIBFA.LIB, 5-24
 - xLIBFP.LIB, 5-25
- Library considerations for
 - floating-point options, 9-13
- Library functions
 - ANSI standards and CTOS
 - implementation, C-19
 - migration, A-22
- Library support
 - for multithread program, 16-2
- LIMITS.H
 - data ranges defined in, D-4
 - numerical ranges for defined constants, D-4
- Line-continuation character (\), 10-6
- LINK description, B-22
- Linker
 - cross references between
 - MLINK and MCBIND, 5-15
 - how to specify options from CL, 5-9
 - MLINK command form, 5-7
 - using, 5-6
- Linker configuration file, 6-24
- Linker options
 - setting in PWB, 3-39
- Linker options that control
 - optimization, 6-24
- Linker response files, 5-12
- LinkerConfig.sys, 6-24
- Linking
 - controlling automatic, 5-9
 - floating-point support, 5-25
 - for real mode with _huge
 - memory, 5-32
 - mixed-language
 - programming, 14-11
 - overview, 1-2
 - PWB extensions, 12-31
 - rules that void with alternate
 - math library, 9-11
 - with floating-point libraries
 - other than the default, 9-14

- Linking minimum-sized run file, 5-24
- Literals
 - using control characters as, 10-8
- LLIBCMT.LIB, 16-2
- Load-time and run time
 - Linking, 16-9
- Local context, 11-16
- Local data
 - allocation in
 - assembly-language procedure, 14-24
- Logical AND operator (&&), 10-25
- Logical OR operator (||), 10-25
- LOGICAL*2 type, 14-35
- LOGICAL*4 type, 14-35
- Long data types
 - byte ordering on various machines, 15-32
- Long double float data type
 - C run-time library support, 9-5
 - size, 15-5
 - support for, 9-1
- longjmp, 5-3
- LONG_MAX, 15-5
- LONG_MIN, 15-5
- Loop optimization
 - disabling unsafe, 6-16
 - performing, 6-15
- Loops, 6-15
- Loop_opt pragma, 6-15

M

- Macros
 - creating trouble-free in _asm blocks, 8-18
 - in-line assembly, 8-5
 - NULL, C-20
 - portability issues, 15-28
 - portable character-classification, 15-20
 - predefined, A-20
 - predefined migration issues, A-20
 - problems with toupper, 15-29
 - PWB macros, 12-6
- MAKE
 - differences from NMAKE, 10-33
- Makefiles
 - creating in PWB, 3-43
 - defined, 10-3
 - examples of, G-1
 - how to create, 10-1
 - how to use NMAKE type in PWB, 5-35
 - notes, 5-34
 - PWB, 5-34
 - two types, 5-34
- MALLOCMODE.Asm
 - changing defaults, 5-31
- Manifest constants
 - NULL, 7-9
 - used in Help files, 11-29

- Mantissa
 - length of for floating-point data types, 9-2
 - storage, 9-3
- MASM
 - advantages over in-line assembler, 14-21
 - invoking automatically from CL, 5-3
 - using, 14-21
- Math coprocessor package, 9-6
- Math intrinsics, 6-10
- Math library functions
 - summary of locations, A-30
- Math packages
 - alternate math, 9-6
 - coprocessor, 9-6
 - emulator, 9-5
 - floating-point, 9-5
- MAXVAL
 - heap size subparameter, 5-28
- MC0.Obj
 - using for primary heap and stack sharing overflow, 5-30
 - using for stack sharing, 5-29
- MCOHEAP.Obj
 - using for primary heap only, 5-30
- MCBIND
 - command form, 5-14
 - compiler option to use, 5-6
 - overview, 1-2
 - sample command form, 5-20
 - using, 5-6
 - using to build for CodeView, 13-4
- MCMEMORY.Obj
 - adding LDTs, 5-31
 - using, 5-32
- Memory
 - _huge, 5-32
 - allocating zero, C-27
 - changing addresses in CodeView, 13-28
 - displaying in CodeView, 13-25
 - handling insufficient memory situations, 15-19
 - managing, 7-1
 - RAM requirements, 2-1
- Memory models
 - advantages and disadvantages of using, 7-5
 - customizing, 7-20
 - default, 5-3
 - how choosing affects default pointer size, 14-12
 - huge, 7-8
 - library support for customized, 7-25
 - limitations on code and data size, 7-8
 - limits, 7-6
 - migration issues, A-11
 - mixing, 7-11
 - mixing rules, 7-13
 - selecting, 7-6
 - standard, 7-5
 - standard with custom equivalents, 7-21
- Menus
 - PWB, 3-8
 - using QuickHelp, 4-7
- Microsoft C Keywords with DLLs, 16-18
- Migrating from BTOC C to CTOS Microsoft C
 - program changes, A-2

Migration

- BTOS C to CTOS, A-1
- case translation issues, A-7
- coding convention issues, A-14
- COMDEF OMF vs. PUBDEF records, A-20
- compiler option issues, A-15
- documented BTOS C function with substitutes, A-4
- documented BTOS C functions without substitutes, A-3
- DS allocation, A-21
- extended keywords, A-13
- function character limits, A-15
- function issues, A-2
- function prototypes, A-14
- function status from BTOS C to CTOS, A-22
- global variable issues, A-20
- header file conversions, A-7
- header file differences: BTOS to CTOS, A-6
- huge memory model, A-11
- initialization issues, A-15
- keyword issues, A-12
- library functions, A-22
- math library function, A-30
- memory model issues, A-11
- mixed-language programming, A-14
- predefined macro issues, A-20
- replacing ctos.h references with ctos.lib.h, A-6
- undocumented BTOS C functions, A-2
- undocumented BTOS C functions with work-arounds, A-5

MIN.OBJ

- defined, 5-22

- Minimally formatted ASCII text file, 11-30

- Mixed-language programming parameter passing, 14-16

- Mixed language calls

- executing, 14-12

- how they work, 14-1

- making, 14-1

- syntax, 14-2

- Mixed-language naming

- convention requirements, 14-4

- Mixed-language programming, 14-1

MLIB

- command form summary, B-28

- command form syntax, 5-26

- command line options, B-28

MLINK

- /NOD described, 5-17

- /PACKC described, 6-26

- automatic operation, 5-6

- command form, 5-7

- command form summary, B-29

- command line syntax, 5-7

- compiler option used to disable automatic linking, 5-3

- config file parameters, 6-25

- controlling automatic linking, 5-9

- default libraries, 5-17

- default libraries when used manually, 5-13

- defaults related to MCBIND fields, 5-12

MLINK (continued)

- how to use, 5-7
 - manual operation, 5-6
 - options correlating to linker
 - config file options, 5-17
 - overview, 1-2
 - using, 5-6
 - using a response file, 5-12
 - using to build for CodeView,
 - 13-4
- MODEL directive**, 8-12
- Modules**
 - naming, 7-27
- Modulus operator (%)**, 10-25
- Monitors**
 - how Help text appears, 11-26
- Multibyte characters**
 - ANSI standard and CTOS
 - implementation, C-5
- Multiplication operator (*)**, 10-25
- Multithread**
 - creating CTOS III
 - multithread applications,
 - 16-1
 - Include files, 16-5
- Multithread C Library**
 - LLIBCMT.LIB, 16-3
- Multithread library compile option /MT**, 16-4
- Multithread program**, 16-1
- Mutithread options**, 16-4

N

- Naming conventions**
 - /Gc option and affects on
 - Pascal programs, 14-13
 - data types for C, FORTRAN,
 - and Pascal, 14-34
 - defaults for languages, 14-33
 - mixed-language identifiers,
 - 14-4
 - with mixed languages, 14-9
- Naming modules and segments**, 7-27
- Near heap memory**
 - controlling placement of in
 - DGroup, 5-27
 - sizing, 5-27
 - three options for placing in
 - DGroup, 5-28
 - using primary area only, 5-30
- Near pointers**
 - how they work, 7-2
- Nested comments**, A-19
- New Features**, 1-11
- Newline character**
 - placing in a description file,
 - 10-9
 - terminating, C-22
- Newline escape sequence**, 10-7

NMAKE

- Description Files, 10-6
- command example, 10-3
- command form summary, B-38
- command line options, 10-28
- differences from MAKE, 10-33
- directive operators, 10-25
- environment variables
 - recognized, B-38
- indentation of command lines, 10-6
- in-line files, 10-30
- line-continuation character (\), 10-6
- managing development
 - projects with, 10-1
- operations sequence, 10-31
- overview, 1-3
- spawning vs. submit files, 5-35
- two methods of executing
 - makefiles, 5-35
 - using with Batch, 5-35

NMAKE description files

- example, creating a library, G-21
- example, creating a library
 - version string, G-4
- example, creating a PWB
 - extension, G-6
- example, creating a run file, G-12
- example, creating Help files, G-2
- example, creating run files, libraries, and a system service, G-30

NMAKE directives

- rules for using, 10-26

NMAKE inference rules

- extension search paths, 10-21
- precedence, 10-22
- predefined, 10-22
- user defined, 10-20

NMAKE macros

- case distinction, 10-15
- environment variable effects, 10-19
- in description files, 10-11
- inherited from environment
 - variables, 10-19
- invoking, 10-13
- precedence among definitions, 10-19
- predefined list, 10-14
- redefining inherited, 10-19
- rules for handling spaces, 10-12
- substitution within, 10-17
- syntax, 10-12
- user-defined, 10-12

NO87 environment variable

- using, 9-15

NOP instruction, 8-5

Normalized value, C-11

Null characters, C-23

NULL definition, 15-17

NULL macro, C-20

NULL pointers, 7-9

- portability, 15-17

Number sign (#), 10-11

Numeric data representation

- across languages, 14-33

Numerical ranges, D-4

Numerical values defined in FLOAT.H, D-6

O

Obj Flavor object modules

 REALHUGE.Obj, 5-32

One's complement arithmetic,
 15-15

Online Help, 4-1

Optimization

 /Ol, 6-15

 from PWB, 6-2

 linker and packing code, 6-28

Open files

 increasing the maximum

 number, 2-15

Operating systems

 compatibility, 2-1

Operations sequence

 NMAKE, 10-31

Operators

 directive for NMAKE, 10-25

 using in `_asm` blocks, 8-9

Optimization

 /G0, /G1, and /G2 described,
 6-21

 /Gc described, 6-30

 /Gd described, 6-29

 /Gr described, 6-30

 /Gs described, 6-18

 /Oa and /Ow described, 6-12

 /Oc and /Og described, 6-20

 /Oe described, 6-18

 /Oe incompatible with /r,
 6-20

 /Oi described, 6-8

 /On described, 6-16

 /Op described, 6-20

 /Or disables /Oc, 6-20

 /Or incompatible with /Og,
 6-20

 /Ot and /Os described, 6-8

Optimization (continued)

 /Ou and /Oy described, 6-23

 /Ox described, 6-22

 /Oz described, 6-16

 achieving consistent

 floating-point results, 6-20

 achieving the smallest code,
 6-23

 aggressive, 6-16

 argument-passing convention,
 6-31

 assume no aliasing, 6-12

 C calling convention, 6-29

 common subexpression

 elimination, 6-5

 constant propagation, 6-6

 constant subexpression

 elimination, 9-10

 controlling from the command
 line, 6-3

 controlling with pragmas, 6-3

 customizing, 6-7

 dead-store elimination, 6-6

 default, 6-5

 default compiler switches,
 6-7

 disabling unsafe loop, 6-16

 disabling with switches and
 pragmas, 6-5

 effects of `_asm` blocks, 8-20

 enabling aggressive, 6-16

 enabling common

 subexpression, 6-20

 enabling far call, 6-25

 for debug compiles, 6-2

 for release compiles, 6-2

 for speed, 6-3

 FORTRAN/Pascal calling
 convention, 6-30

- Optimization (continued)
 - function calling conventions, 6-29
 - function prolog and epilog, 6-23
 - generating intrinsic function, 6-8
 - global incompatibility with setjmp and longjmp, 5-3
 - global register allocation, 6-18
 - linker, 6-24
 - linker and enabling far call, 6-25
 - linker options that control, 6-24
 - list of compiler switches, 6-7
 - loops, 6-3
 - maximum efficiency, 6-22
 - performing loop, 6-15
 - register calling convention, 6-30
 - removing stack probes, 6-18
 - specifying compiler, 6-1
 - speed or size, 6-8
 - three ways by the compiler, 6-1
 - troubleshooting, 6-14
 - types inhibited by in-line assembly code, 8-21
- Optimization
 - using the 80186, 80188, or 80286 processor, 6-21
 - without aliasing, 6-3
- Optimizing C programs, 6-1
- Optimizing program size, 6-8
- Optimizing program speed, 6-8
- Order and alignment issues
 - structures, 15-8
 - unions, 15-10
- Output redirection, 5-39

- Overlays
 - generating, 6-24
 - how to create, 5-4
 - incompatibilities, 5-5
 - migration, A-21
 - producing compatible code, A-21
- Overview
 - Browser, 1-7
 - CodeView, 1-6
 - compiler and linker, 1-2
 - CTOS Microsoft C, 1-1
 - Environment Service, 1-11
 - Help utilities, 1-9
 - MLIB, 1-10
 - NMAKE, 1-3
 - PWB, 1-5
 - Task Management Service, 1-7

P

- PackCode, 6-28
- Packing code, 6-28
- Parameter passing
 - defaults for mixed languages, 14-11
 - mixed-language programming, 14-9
- Parameters
 - accessing in assembly, 14-27
 - mixed-language programming, 14-43
 - passing by reference or value in mixed languages, 14-16
 - passing defaults for mixed languages, 14-11
 - passing requirements, 14-9

- Parameters (continued)
 - passing using far reference, 14-10
 - passing using near reference, 14-10
 - passing using variable values, 14-10
 - three methods for passing, 14-9
- pArg, 12-21
- Partition
 - creating a QuickHelp, 4-25
- Pascal
 - calling a Pascal program, 14-1
 - calling conventions, 14-7
 - conventions, 14-33
 - data type naming
 - conventions, 14-34
 - far memory and far reference, 14-10
 - LSTRING not compatible
 - with C, 14-37
 - naming conventions, 14-4
 - string format, 14-36
- Passing addresses of common blocks, 14-42
- Passing parameters
 - mixed-language agreement on
 - reference or value, 14-16
 - using variable values, 14-10
 - with far reference, 14-10
 - with near reference, 14-10
- perror function, 5-40
- perror function messages, C-26
- Pointer arithmetic
 - in huge models, 7-9
- Pointer conversion
 - %hp, C-25
 - %lp, C-25
 - %p, C-25
- Pointer subtraction
 - ANSI standards and CTOS implementation, C-12
- Pointer values
 - printing, C-25
 - reading, C-25
- Pointers
 - advantages of based, 7-30
 - advantages of near and far, 7-30
 - and aliasing optimization, 6-12
 - ANSI standards and CTOS implementation, C-11
 - based on a pointer, 7-36
 - based on a segment constant, 7-33
 - based on a segment variable, 7-34
 - based on a self segment, 7-38
 - based on void, 7-38
 - based problems, 7-15
 - casting, C-11
 - casting far to near, 7-13
 - casting portability, 15-16
 - checking for undefined in CodeView, 13-30
 - compiler issues, 6-12
 - conversions, 7-15
 - default sizes, 15-19
 - determining size, 15-16
 - far, 7-3
 - how memory model affects
 - size, 14-12
 - huge, 7-4
 - huge and migration issues, A-11
 - keywords, 7-17

Pointers (continued)

- near, 7-2
- null, 15-17
- NULL data, 7-9
- null sizing issues, 7-9
- passing as arguments in
 - multiple-segment programs, 7-23
- passing as arguments to
 - functions, 7-15
- passing in mixed-language programming, 14-41
- portability, 15-15
- problems to avoid, 7-13
- relationship to 64K segments, 7-1
- rules for declaring variables
 - with keywords, 7-17
- setting all to one size with a standard memory model, 7-5
- size of generic in bits, 15-18
- sizes of generic in a table, 15-18
- sizing code in a customized memory model, 7-22
- sizing data, 7-22
- subtraction, C-12
- subtraction of char pointers in
 - character arrays, 15-18
- types pushed on the stack, 6-31
- types supported, 7-2

Portability

- address space issues, 15-19
- arithmetic modes, 15-14
- bitwise right-shift operations, 15-25

Portability (continued)

- byte ordering in words, 15-10
- byte ordering issues, 15-31
- case sensitivity, 15-26
- char data type, 15-3, 15-24
- character classifications in
 - character sets, 15-20
- character set case translation issues, 15-21
- character set issues, 15-20
- compiler issues and
 - assumptions, 15-22
- CTOS Microsoft C concerns, 15-31
- data file issues, 15-30
- data type sizing issues, 15-2
- environment issues, 15-30
- expression evaluation order, 15-27
- float data types, 15-5
- function and macro argument issues, 15-28
- functions with a variable
 - number of arguments, 15-27
- hardware issues, 15-2
- identifier length and case, 15-25
- int and short int data types, 15-3
- issues involving `_near`, `_far`, `_huge`, and `_based` keywords, 7-11
- list of nonportable keywords, 15-31
- null pointer issues, 15-17
- one's complement arithmetic issues, 15-14

- Portability (continued)
 - pointer casting issues, 15-16
 - pointer issues, 15-15
 - pointer size issues, 15-16
 - pointer subtraction issues, 15-16
 - promotion from shorter types, 15-22
 - promotion of data types, 15-22
 - register issues, 15-26
 - sign extension issues, 15-22
 - storage alignment issues, 15-12
 - storage order and alignment issues, 15-7
 - structure bit fields, 15-12
 - structure bit fields order of assignment, 15-13
 - structure issues, 15-8
 - two's complement arithmetic issues, 15-14
 - union issues, 15-10
 - word byte ordering, 15-10
- Portable programs
 - writing, 15-1
- Post-optimization, 6-26
- Pragma
 - ANSI standards and CTOS implementation, C-18
 - check_stack, 6-18
 - controlling optimization, 6-3
 - defined, 6-3
 - how to use, 6-4
 - intrinsic, 6-10
 - list of control optimization, 6-3
- Pragma (continued)
 - loop_opt, 6-15
 - migration, A-13
 - optimize with e, 15-26
 - used to maintain correct
 - compiles with setjmp and longjmp, 5-3
- Preprocessing directives
 - ANSI standards and CTOS implementation, C-16
- Preserving register values in assembly, 14-25
- printf format specifiers, E-1
- Printing
 - from CodeView, 13-31
 - from QuickHelp, 4-24
- Printing pointer values, C-25
- Processor arithmetic mode, 15-14
- Processor clock cycles for calling sequence, 6-27
- Processor compatibility, 6-22
- Program limits at run time, D-3
- Program size
 - optimizing, 6-8
- Program speed
 - optimizing, 6-8
- Programmer's WorkBench, 3-1
 - notes, 5-36
 - with mixed languages, 14-1
- Programs
 - CTOS Microsoft C portability concerns, 15-31
 - optimizing C, 6-1
 - writing portable, 15-1
- Promotion
 - from shorter data types, 15-22
 - of Floating-point types, 9-4

- Prototypes
 - declaring external routines for program documentation, 14-13
 - functions and address lengths, 14-13
- Pseudofiles
 - as used in PWB, 12-2
- Pseudoinstruction
 - in-line assembly, 8-7
- Pseudotargets
 - .IGNORE:, 10-27
 - .SILENT, 10-27
 - .SUFFIXES, 10-27
- ptrdiff_t, C-12
- PUBDEF
 - in received object streams, A-20
 - versus COMDEF OMF record, A-20
- PUBLIC directive, 14-23
- PWB
 - assigning keystrokes, 12-4
 - assigning multiple keystrokes to one function, 12-4
 - Browse menu, 3-13
 - building programs, 3-47
 - building programs from multiple source files, 3-48
 - changing Editor settings, 3-28
 - changing keystrokes, 12-4
 - choosing menu commands, 3-14
 - closing menus, 3-14
 - compiling, linking, and running programs, 3-30
 - creating a build option, 3-41
 - creating and selecting makefiles, 3-43

- PWB (continued)
 - creating macros, 3-26
 - customizing the Editor, 3-27
 - debugging programs, 3-53
 - debugging with error messages, 3-54
 - defining a block, 3-21
 - dialog box Help, 3-18
 - dialog boxes, 3-15
 - disabling keystrokes, 12-5
 - Edit menu, 3-10
 - editing TOOLS.INI, 12-3
 - editor settings, 12-2
 - File menu, 3-9
 - four methods to customize, 12-1
 - getting Help, 3-18
 - Help menu, 3-18
 - limitations on keystroke assignments, 12-5
 - Make menu, 3-11
 - menus, 3-8
 - modifying keyboard assignments, 3-29
 - moving around in a source file, 3-20
 - Options menu, 3-12
 - overview, 1-5
 - overview of menu commands, 3-8
 - overview of setting compiler and linker options, 3-31
 - parts of a screen, 3-6
 - pseudofiles, 12-2
 - quick reference to setting compiler and linker options, 3-52
 - Run Application, 3-48

PWB (continued)

- Run menu, 3-12
- running programs, 3-48
- sample program, 3-34
- saving compiler and linker
 - options as a build option, 3-41
- saving source files, 3-35
- Search menu, 3-11
- searching for and changing
 - text, 3-24
- selecting a build option, 3-46
- setting anchors, 3-23
- setting bookmarks, 3-22
- setting compiler options, 3-35
- setting linker options, 3-39
- setting switches, 12-2
- shaded commands, 3-15
- starting, 3-2
- starting and running
 - CodeView, 3-64
- using, 3-1
- using advanced editor
 - features, 3-30
- using shortcut keys, 3-15
- using the Browser, 3-55
- using the Editor, 3-19
- using windows, 3-4
- View menu, 3-10

PWB extensions

- arg, 12-21
- argData, 12-21
- argType, 12-21
- assigning keystrokes to, 12-33
- building and using, 12-28
- cmdTable, 12-16
- compiling, 12-29
- debugging, 12-35
- fMeta, 12-21

PWB extensions (continued)

- how they differ from run files, 12-13
- linking, 12-31
- loading, 12-32
- mandatory elements, 12-16
- pArg, 12-21
- RADIX10, 12-19
- reading a line from a file, 12-24
- receiving parameters, 12-21
- sample Center.C, 12-14
- summary of functions, 12-24
- swiDesc, 12-18
- swiTable, 12-18
- SWI_BOOLEAN, 12-18
- SWI_NUMERIC, 12-19
- SWI_SPECIAL, 12-18
- using to get a file handle, 12-23
- writing and building, 12-13
- writing function prototypes, 12-16
- writing functions D, 12-20
- writing the command table, 12-16
- writing the initializing
 - function, 12-20
- writing the switch table, 12-18

PWB macros

- appending commands to, 12-13
- argument syntax, 12-8
- arguments, 12-9
- conditional operators, 12-10
- definitions, 12-7
- macro responses, 12-8
- macros invoking macros, 12-8

PWB macros (continued)
 passing arguments to
 functions, 12-7
 procedure to create, 12-11
 procedure to record, 12-12
 procedure to save
 permanently, 12-12
 recording, 12-12
 syntax, 12-6
 temporary, 12-11
 working with literal text, 12-6
 writing, 12-6
PWBRMAKE
 command form summary,
 B-41
 command line options, B-41
 using, 3-55

Q

Qualifiers
 ANSI standards and CTOS
 implementation, C-15
Question mark (?), 10-7
QuickHelp
 accessing error information,
 4-23
 command form, 4-2
 command form summary,
 B-43
 command line options, B-43
 copying and pasting, 4-18
 creating a partition, 4-25
 cross references, 11-27
 dialog boxes, 4-15
 displaying keyword
 information, 4-21
 dot commands, 11-23
 exiting, 4-4

QuickHelp (continued)
 formatting flags, 11-26
 navigating through screens,
 4-15
 running, 4-2
 starting, 4-2
 structure, 4-5
 using, 4-1
 using hyperlinks, 4-17
 using menus, 4-7
 using pull-down menus and
 dialog boxes, 4-7
QuickHelp format text file,
 11-22
Quoted file names, C-17

R

RAM
 system requirements, 2-1
Range of char values
 ANSI standard and CTOS
 implementation, C-7
Range of integer values
 ANSI standard and CTOS
 implementation, C-8
Ranges
 reading, C-25
Re-entrant code with DLLs,
 16-16
Reading pointer values, C-25
Reading ranges, C-25
REALHUGE.Obj
 defined, 5-22
Records
 mixed-language
 programming, 14-39
Redirection
 and starting CodeView, 13-32
 input, 5-39
 output, 5-39

- Register calling convention, 6-29
 - Register candidates for
 - argument types, 6-31
 - Register declarations, 15-26
 - Register variable storage, 8-20
 - Register variables
 - portability, 15-26
 - Registers
 - ANSI standards and CTOS
 - implementation, C-12
 - assembly conventions for
 - simple return values, 14-30
 - availability, C-12
 - change values in CodeView, 13-28
 - displaying in CodeView, 13-27
 - enabling global allocation, 6-18
 - handling variables in
 - CodeView, 13-32
 - preserving in assembly, 14-31
 - preserving values in assembly
 - procedure, 14-25
 - preserving with /Ou, 6-23
 - returning ES:BX, 5-37
 - Remainders
 - ANSI standard and CTOS
 - implementation, C-9
 - Rename function, C-25
 - Renaming a file with an existing name, C-25
 - Return values
 - assembly register conventions
 - for simple, 14-30
 - from assembly-language
 - procedure, 14-30
 - Returning a value from
 - assembly, 14-30
 - Rich text format, 11-31
 - Rich text format RTF, 11-4
 - Right shift
 - ANSI standard and CTOS
 - implementation, C-9
 - bitwise operations, 15-24
 - Right shift operator (>>), 10-25
 - Routines
 - mixed-language call
 - equivalents, 14-3
 - RTF, 11-31
 - Run files
 - _huge memory support for
 - real mode, 5-21
 - creating a GDT protected
 - with MCBIND, 5-24
 - creating a minimum sized
 - with MLINK, 5-25
 - creating typical, 5-19
 - creating with MLINK, 5-2
 - linking minimum sized, 5-24
 - linking minimum sized with
 - MCBIND, 5-24
 - sample CL command form
 - that chains to MLINK, 5-10
 - unpredictable PWB build
 - behavior, 5-34
 - version 8, 5-17
 - Run-time library support of
 - long double types, 9-5
 - Run-time program limits, D-3
- ## S
- scanf format specifiers, E-1
 - SCHAR_MAX, 15-4
 - SCHAR_MIN, 15-4
 - Segment overrides
 - in-line assembly, 8-5

- Segments
 - adding more far heap, 5-32
 - code and data, 7-1
 - far heap, 5-32
 - LDT-based, 5-31
 - naming, 7-27
 - naming conventions for code and data, 7-27
 - setting up, 7-23
 - specifying code and data, 7-29
- Sequence point, 15-27
- setjmp, 5-3
- Setting the data threshold, 7-26
- Setting up the assembly
 - procedure, 14-23
- Short data types
 - byte ordering on various machines, 15-32
- Short int data type
 - size, 15-3
- SHRT_MAX, 15-4
- SHRT_MIN, 15-4
- Sign extension
 - portability issues, 15-22
- Signed bitwise operations
 - ANSI standard and CTOS implementation, C-9
- Significant characters without external linkage
 - ANSI standard and CTOS implementation, C-4
- Size
 - basic data types, 15-2
 - char data type, 15-3
 - default pointers, 15-19
 - determining pointer, 15-16
 - determining with the sizeof operator, 15-2
 - double float data type, 15-5
- Size (continued)
 - float data type, 15-5
 - int data type, 15-3
 - long double float data type, 15-5
 - of generic pointers in bits, 15-18
 - pointers and memory models, 14-12
 - short int data type, 15-3
- sizeof operator
 - example of using, 15-10
 - using, 15-2
- Size_t type, C-11
- Slash (/), 11-6
- Square brackets ([]), 10-26
- Stack
 - adjusting, 14-8
 - adjustment convention, 6-32
 - data types pushed onto, 6-31
 - how languages push
 - parameters onto, 14-8
 - mixed-language calls and responsibility for adjusting, 14-7
 - NDP , 6-32
 - passing parameters in
 - mixed-language programming, 14-8
 - setting up the frame in
 - assembly, 14-24
 - sharing memory with near heap, 5-29
 - sharing with near heap
 - overflow, 5-30
- Stack frame
 - example, 14-28
 - how assembly functions use, 14-27
 - removing generation code with /Oy, 6-24

- Stack overflow, 6-18
- Stack probes
 - removing, 6-18
- Stand-alone DLL files, 16-21
- Stand-alone libraries, 16-11
- Startup object modules
 - combining MC0.Obj or MC0Heap with stack and heap specifications, 5-28
- Startup object modules
 - architecture and startup object modules, 5-27
 - defined, 5-21
 - Flavor Object Modules, 5-22
 - MC0HEAP.Obj and near heap, 5-30
 - requirements, 5-20
 - using MC0.Obj for stack sharing, 5-29
- Statements
 - ANSI standards and CTOS implementation, C-15
- STD instruction, 8-14
- Storage alignment
 - of data types, 15-12
- Storage order and alignment
 - portability, 15-7
- Streams
 - increasing the number of, 2-16
- strerror function, C-28
- STRING.h
 - contents, 7-14
- Strings
 - C format, 14-35
 - FORTTRAN format, 14-36
 - handled by C, FORTRAN, and Pascal, 14-35
 - Pascal format, 14-36
 - passing C to FORTRAN and Pascal, 14-36
 - passing Pascal to C, 14-37
- Strong typing, 6-31
- Structures
 - alignment of members, A-19
 - ANSI standards and CTOS implementation, C-13
 - bit fields in, 15-12
 - bit fields order of assignment, 15-13
 - displaying in CodeView, 13-21
 - mixed-language programming, 14-39
 - nameless, 5-40
 - non-portable code example, 15-8
 - order and alignment issues, 15-8
 - passing across mixed-language interface, 14-40
 - portability assumptions, 15-8
 - pushed on the stack, 6-31
 - reading and writing, 15-12
 - size and alignment of bit fields, 15-13
 - specifying CTOS system, A-19
 - storage methods of C, FORTRAN, and Pascal, 14-39
- Subexpression elimination, 6-5
- Subexpression optimization
 - enabling common, 6-20
- Subtraction
 - pointer portability issues, 15-16
- Subtraction operator (-), 10-25
- Summary command button, 4-22
- Switch statements
 - limits, C-15
- Switches
 - setting in PWB, 12-2

Use alternate math compiler
option, 9-13

User-defined types
mixed-language
programming, 14-39
USHRT_MAX, 15-4

V

Values
range of integer, C-8
Variables
based on a segment constant,
7-33
changing in CodeView, 13-20
declaring as floating-point
data types, 9-2
declaring based, 7-32
displaying in CodeView, 13-19
floating-point precision and
storage, 9-4
promoting floating-point, 9-4
register, 15-26
using based, 7-30
Volume name
size, C-24

W

WhenLoaded, 12-20
Wide characters
ANSI standard and CTOS
implementation, C-6
Wildcard characters, 10-7
Word
byte ordering, 15-10
Writing a multithread program,
16-7
Writing Assembly-language
procedures, 14-22
Writing portable programs
hardware assumptions, 15-2
programming assumptions,
15-1

X

xLIBCP.LIB
defined, 5-23
xLIBFP.LIB
defined, 5-24

Z

Zero-length files, C-24

Syntax rules for keywords,
14-14
system function, C-28

T

Targets
 defined, 10-2
 how NMAKE updates, 10-32
 specifying in multiple
 description blocks, 10-10
Task Management Service
 installing hard disk, 2-11
 installing memory, 3-2
 overview, 1-7
 starting, 13-7
Text files
 truncation, C-23
Text mode, 5-41
Thread stacks, 16-8
Threads, 16-2
Time
 default, C-19
Time zone, C-30
time_t, A-5
TOOLS.INI
 customizing CodeView, 13-33
 editing in PWB, 12-3
 for CodeView, 13-6
 for PWB, 12-2
Topic text, 11-2
toupper
 problems implementing as a
 macro, 15-29
Translation
 ANSI standards and CTOS
 implementation, C-2

Truncation of floating-point
 values
 ANSI standard and CTOS
 implementation, C-11
Two's complement arithmetic,
15-15

U

UCHAR_MAX, 15-4
UINT_MAX, 15-5
Underflow of floating-point
 values, C-22
Underscore (_), 14-4
Undocumented BTOS C
 functions
 migration issues, A-2
Unformatted ASCII text file,
11-4
Unions
 ANSI standards and CTOS
 implementation, C-13
 nameless, 5-40
 non-portable code example,
 15-10
 order and alignment issues,
 15-10
 portability assumptions, 15-8
 pushed on the stack, 6-31
Unrepresented character
 constants
 ANSI standard and CTOS
 implementation, C-6
Unsafe loop optimizations
 disabling, 6-16
Up command button, 4-17

**This Page is Blank
Do Not Print**



43931609-000